Complete Networking Reference

Wi-Fi Adapters · IP Addressing · Windows & Linux Commands
Routing · DNS · DHCP · Embedded Networking · Practical Scenarios

Veritas Technical Expertise LLC · June 2026
Generated by Perplexity Computer

Networking Reference — Ty's Field Manual

Audience: Advanced electronics/cybersecurity technician refreshing and filling gaps.

Format: No fluff. Full explanations. Windows (cmd/PowerShell) + Linux commands throughout. Embedded/microcontroller context where relevant.

Table of Contents

- [Part 1: Foundational Concepts](#part-1-foundational-concepts)
- IP Addressing, Subnet Masks, IPv6, Private Ranges, Loopback
- MAC Addresses, ARP, Layer 2 vs Layer 3
- OSI Model (practical)
- DNS (records, resolution chain, hosts file)
- DHCP (DORA, leases, reservations)
- Gateways, Routing, NAT
- [Part 2: Network Adapter Settings](#part-2-network-adapter-settings)
- IP, Subnet, Gateway, DNS, Search Domains, Proxy
- MAC Randomization, iCloud Private Relay
- Wi-Fi: Bands, Channels, Width, Power
- RTS/CTS, Fragmentation, Beacon, DTIM, WMM
- 802.11 protocols, BSSID/SSID, Hidden networks
- WPA2/WPA3, PMF, TKIP vs CCMP
- [Part 3: Windows Networking Commands](#part-3-windows-networking--complete-command-reference)
- ipconfig (all flags), netsh (ip + wlan + firewall)
- PowerShell (Get-NetAdapter, Test-NetConnection, Resolve-DnsName, etc.)
- ping, tracert, pathping, nslookup, netstat, arp, route
- [Part 4: Linux Networking Commands](#part-4-linux-networking--complete-command-reference)
- ip addr/link/route/neigh, iw, iwconfig, nmcli, ifconfig
- ss/netstat, ping, traceroute, mtr, dig
- iptables/nftables, network config files, /etc/resolv.conf
- [Part 5: Microcontroller/Embedded Networking](#part-5-microcontroller--embedded-networking)
- ESP32/ESP8266 Wi-Fi (Arduino), static IP, scanning
- TCP vs UDP, HTTP vs MQTT vs raw TCP
- 802.11 association at packet level, DHCP on embedded
- [Part 6: Practical Scenarios](#part-6-practical-scenarios)
- Static IP setup, no-internet diagnosis, quick two-machine network
- Proxy settings, network discovery, Wi-Fi drops diagnosis

PART 1: FOUNDATIONAL CONCEPTS

IP Addressing

IPv4 Structure — Octets, Binary, Why 255 is the Max

An IPv4 address is a 32-bit number. The "dotted decimal" notation (e.g., `192.168.1.10`) is just a human-readable way to represent four 8-bit groups called **octets**.

Each octet is 8 binary digits (bits). The maximum value of 8 bits is `11111111` in binary, which equals 255 in decimal. That's why 255 is the ceiling — you can't store a number larger than 255 in 8 bits without overflowing into the next byte.

Binary breakdown of `192.168.1.10`:

Octet	Decimal	Binary
1	192	1100 0000
2	168	1010 1000
3	1	0000 0001
4	10	0000 1010

Binary-to-decimal position values (powers of 2, right to left):

```
128  64  32  16   8   4   2   1
  1   1   0   0   0   0   0   0  = 128 + 64 = 192
```

The full 32-bit IPv4 space gives 2^32 = 4,294,967,296 possible addresses. This sounded enormous in the 1980s. It is not.

Subnet Masks — What They Actually Do

A subnet mask is a 32-bit number that defines which portion of an IP address identifies the **network** and which portion identifies the **host** within that network.

The mask works by having all the **network bits set to 1** and all the **host bits set to 0**.

Example: `255.255.255.0` in binary:

```
11111111.11111111.11111111.00000000
```

The first 24 bits are the network portion. The last 8 bits are the host portion.

Applied to IP `192.168.1.10`:

```
IP:   11000000.10101000.00000001.00001010
Mask: 11111111.11111111.11111111.00000000
```

The network is `192.168.1.0`. The host is `.10`.

CIDR notation (`/24`) just counts the number of consecutive 1-bits in the mask from the left. `/24` = 24 ones = `255.255.255.0`. `/16` = `255.255.0.0`. `/8` = `255.0.0.0`.

Common masks:

CIDR	Subnet Mask	Hosts Available
/8	255.0.0.0	16,777,214
/16	255.255.0.0	65,534
/24	255.255.255.0	254
/25	255.255.255.128	126
/26	255.255.255.192	62
/27	255.255.255.224	30
/28	255.255.255.240	14
/29	255.255.255.248	6
/30	255.255.255.252	2

Hosts available = 2^(host bits) - 2 (subtract network address and broadcast address).

How a Device Decides: Local or Router?

When your device wants to send a packet to an IP address, it must decide: is this target on **my local network segment** or do I need to send it to the **default gateway** (router)?

It does this using a **bitwise AND operation**:

1. AND the **source IP** with the **subnet mask** → get source network address
2. AND the **destination IP** with the **subnet mask** → get destination network address
3. If they match → destination is local → send directly via ARP
4. If they don't match → destination is remote → send to default gateway

Example:

- Your IP: `192.168.1.50`, Mask: `255.255.255.0`
- Target: `192.168.1.200`

```
192.168.1.50  AND  255.255.255.0  = 192.168.1.0
192.168.1.200 AND  255.255.255.0  = 192.168.1.0
```

Same network → send directly.

- Target: `10.0.0.5`

```
10.0.0.5 AND 255.255.255.0 = 10.0.0.0
```

Different from `192.168.1.0` → send to router.

This is why setting the wrong subnet mask breaks things in subtle ways: the device will try to ARP for things it should be routing, or route things it should be ARPing for.

Network Address, Broadcast Address, Usable Host Range

Given any IP/CIDR block, you can derive all three:

Example: `192.168.1.0/24`

- **Network address:** Set all host bits to 0 → `192.168.1.0` (not assignable to a host)
- **Broadcast address:** Set all host bits to 1 → `192.168.1.255` (sends to all hosts on segment)
- **Usable range:** Everything between → `192.168.1.1` through `192.168.1.254` (254 hosts)

Example: `10.10.5.0/28`

A /28 mask is `255.255.255.240` = `11111111.11111111.11111111.11110000`

- Host bits: 4 bits → 2^4 = 16 addresses, minus 2 = **14 usable hosts**
- Network address: `10.10.5.0`
- Broadcast: `10.10.5.15`
- Usable: `10.10.5.1` – `10.10.5.14`

Arbitrary example: `172.16.4.130/26`

/26 mask = `255.255.255.192` = `...11000000`

The fourth octet host portion: `130` in binary = `10 000010`. The /26 boundary falls at multiples of 64.

- 130 falls in the range 128–191 (64*2=128 start, +63=191 end)
- Network: `172.16.4.128`
- Broadcast: `172.16.4.191`
- Usable: `172.16.4.129` – `172.16.4.190`

IPv6 — Structure, Why It Exists, How It Differs

IPv4 gives 2^32 addresses. IPv6 gives **2^128** — approximately 340 undecillion (3.4 × 10^38). The reason IPv6 exists is exhaustion: IANA gave out the last IPv4 /8 blocks in 2011.

IPv6 address structure:

128 bits written as 8 groups of 4 hex digits separated by colons:

```
2001:0db8:85a3:0000:0000:8a2e:0370:7334
```

Abbreviation rules:

1. Leading zeros in each group can be dropped: `0db8` → `db8`, `0000` → `0`
2. One consecutive sequence of all-zero groups can be replaced with `::` — but only once per address

```
2001:db8:85a3::8a2e:370:7334
```

Key IPv6 differences from IPv4:

Feature	IPv4	IPv6
Address size	32 bits	128 bits
Header size	Variable (20–60 bytes)	Fixed 40 bytes
Broadcast	Yes	No (uses multicast)
ARP	Yes	Replaced by NDP (Neighbor Discovery Protocol)
IPSec	Optional	Built into spec (though not always enforced)
DHCP	Used for addressing	SLAAC can self-configure without DHCP
Fragmentation	Routers can fragment	Only source host fragments
Checksum	In header	Removed (offloaded to L4)
NAT	Nearly universal	Unnecessary by design

Address types:

- **Global Unicast (GUA):** `2000::/3` — publicly routable, starts with `2` or `3`
- **Link-Local:** `fe80::/10` — automatically assigned to every IPv6 interface, only valid on the local link (never routed). Used for neighbor discovery, router advertisements. Even if you have no IPv6 internet, your interfaces have `fe80::` addresses.
- **Unique Local:** `fc00::/7` — equivalent of RFC 1918 private space, not routed on the internet
- **Loopback:** `::1` — equivalent of `127.0.0.1`
- **Multicast:** `ff00::/8` — one-to-many, replaces broadcast

SLAAC (Stateless Address Autoconfiguration): A host can generate its own global unicast address by taking the network prefix from a Router Advertisement and appending a host identifier derived from the MAC

address (EUI-64) or a random value (privacy extensions). No DHCP server required.

Private vs Public IP Ranges (RFC 1918)

RFC 1918 defines three blocks reserved for private use — they are never routed on the public internet:

Block	CIDR	Range	Common Use
10.x.x.x	10.0.0.0/8	10.0.0.0 – 10.255.255.255	Large enterprise, ISP CGN
172.16–31.x.x	172.16.0.0/12	172.16.0.0 – 172.31.255.255	Mid-size networks
192.168.x.x	192.168.0.0/16	192.168.0.0 – 192.168.255.255	Home/small office routers

Also note:

- `100.64.0.0/10` — RFC 6598, Shared Address Space, used by ISPs for Carrier-Grade NAT (CGN)
- `169.254.0.0/16` — RFC 3927, APIPA (Automatic Private IP Addressing) — assigned by Windows/Linux when DHCP fails
- `0.0.0.0` — "this host on this network," used in routing to mean "default" or "all addresses"

Why do they exist? NAT (Network Address Translation) allows thousands of devices behind a single public IP to share that IP when accessing the internet. Private IPs are reused in every home and office in the world. Without NAT + private ranges, IPv4 would have run out far sooner.

Loopback — What It Is and Why

IPv4: `127.0.0.0/8` — the entire /8 block is loopback, though `127.0.0.1` is what's used in practice. **IPv6:** `::1`

The loopback interface (called `lo` on Linux, `Loopback Adapter` on Windows) is a virtual interface that never sends traffic to any physical medium. Packets addressed to `127.0.0.1` stay entirely within the kernel's network stack.

Why it matters:

- Testing network software without a physical network ("does my server listen on port 80?")
- Inter-process communication — many services bind to `127.0.0.1` so only local processes can reach them (more secure than binding to `0.0.0.0`)
- The OS itself uses it: `localhost` resolves to `127.0.0.1` by default via the hosts file
- If you can't ping `127.0.0.1`, the TCP/IP stack itself is broken — this is Layer 3 dead on arrival

---

MAC Addresses

Structure: OUI + Device ID

A MAC (Media Access Control) address is a **48-bit** identifier burned into network interface hardware at manufacture time. Written as 6 groups of 2 hex digits:

```
AA:BB:CC:DD:EE:FF
```

Structure:

- **Bytes 1–3 (OUI — Organizationally Unique Identifier):** Assigned by IEEE to the manufacturer. `00:1A:2B` = specific vendor. You can look up any OUI at `https://regauth.standards.ieee.org`
- **Bytes 4–6 (Device ID / NIC-specific):** Manufacturer-assigned, supposed to be unique per device

Bit flags in the first byte:

- **Bit 0 (LSB):** Unicast (0) vs Multicast (1). `FF:FF:FF:FF:FF:FF` (all 1s) = broadcast
- **Bit 1:** Globally administered (0, factory-burned) vs Locally administered (1, software-set/randomized)

When you see MAC `02:xx:xx:xx:xx:xx` — bit 1 of the first byte is set → locally administered → likely spoofed or randomized.

ARP — How IP Maps to MAC

ARP (Address Resolution Protocol) operates at Layer 2/3 boundary. It answers: **"Who has IP 192.168.1.1? Tell me your MAC."**

The full process:

1. Device A wants to send a packet to `192.168.1.1` (determined to be local via subnet ANDing)
2. Device A checks its **ARP cache** (`arp -a` on Windows/Linux) — is there already an entry?
3. If not: Device A broadcasts an **ARP Request** to `FF:FF:FF:FF:FF:FF` (everyone on segment): "Who has 192.168.1.1?"
4. Device with IP `192.168.1.1` sends a **unicast ARP Reply** back: "192.168.1.1 is at AA:BB:CC:DD:EE:FF"
5. Device A caches this mapping and uses the MAC as the Layer 2 destination for the frame

ARP entries expire (typically 60–300 seconds on most OSes) and are re-resolved. This is why changes propagate — eventually old entries time out.

Gratuitous ARP: A device announces its own IP→MAC mapping without being asked. Used when an interface comes up or an IP changes. Also used in failover clusters to update all neighbor caches immediately.

ARP cache poisoning: An attacker sends fake ARP replies claiming their MAC maps to the gateway's IP. Now all traffic intended for the gateway goes to the attacker instead — classic MITM attack.

Why MAC at Layer 2, IP at Layer 3

This distinction is architectural. The OSI model intentionally separates concerns:

- **Layer 2 (Data Link):** Gets a frame from point A to point B on the **same physical segment**. Uses MAC addresses. A switch operates here — it knows which MAC is on which port and forwards accordingly. MACs are flat (no hierarchy), not routable.
- **Layer 3 (Network):** Gets a packet from any source to any destination across **multiple networks**. Uses IP addresses. Routers operate here. IPs are hierarchical (network + host portion), enabling efficient routing table lookups.

When a packet travels across the internet, the **IP addresses stay constant** (source and destination), but the **MAC addresses change at every hop**. Each router strips the incoming frame, makes a routing decision, and re-encapsulates the packet in a new frame with its own MAC as source and the next-hop MAC as destination.

Analogy: IP is like a postal address (city, street, number). MAC is like the handle on a specific door on a specific floor of an apartment building — only useful locally.

MAC Spoofing — What It Does and Why

Since the MAC is the Layer 2 identity, changing it affects how your device appears on the local network:

What it does:

- Your device sends frames with a different source MAC
- The switch learns the new MAC on your port
- ARP replies and DHCP requests use the spoofed MAC
- The router/DHCP server sees you as a different device

Why you'd do it:

1. **Privacy** — prevent tracking across networks (this is what iOS/Android MAC randomization does per-SSID)
2. **Bypass MAC filtering** — some networks only allow known MACs; spoofing a permitted MAC bypasses this (trivially easy; MAC filtering is weak security)
3. **DHCP reservation bypass or takeover** — if a device has a DHCP reservation tied to its MAC, you can claim that address
4. **Clone a device** — ISPs sometimes bind cable modem service to the MAC of the first device seen; a router that clones the modem's MAC can take over the session

Windows (PowerShell):

```
# Check current MAC
Get-NetAdapter | Select-Object Name, MacAddress

# Change via Device Manager > Advanced > Locally Administered Address
# Or via registry: HKLM\SYSTEM\CurrentControlSet\Control\Class\{4d36e972-...}\000X
# Value: "NetworkAddress" = "AABBCCDDEEFF" (no colons)
```

Linux:

```
sudo ip link set dev eth0 down
sudo ip link set dev eth0 address 02:11:22:33:44:55
sudo ip link set dev eth0 up
# Or with macchanger:
sudo macchanger -r eth0    # random MAC
sudo macchanger -m 02:11:22:33:44:55 eth0
```

OSI Model — Practical, Not Theoretical

The OSI model is a 7-layer abstraction of network communication. In practice, TCP/IP only really uses 4 layers, but OSI helps diagnose and understand where things break.

The Layers, What They Actually Are

Layer 1 — Physical

- Raw bits on wire, fiber, or RF spectrum
- Voltages, light pulses, radio waves
- **Real-world:** Ethernet cables, fiber, coax, Wi-Fi radio, USB-to-Ethernet adapters
- **Breaks when:** Cable is cut/damaged, connector is loose, signal is too weak (too long a run without repeater), RF interference, bad transceiver
- **Diagnose:** Physical inspection, cable tester, check link light on NIC/switch, check RSSI/SNR on Wi-Fi

Layer 2 — Data Link

- Frame construction: adds MAC source/destination, frame check sequence (CRC)
- Splits into two sublayers: MAC (media access) and LLC (logical link control)
- **Real-world:** Ethernet frames, Wi-Fi frames (802.11), PPP, VLAN tags (802.1Q)
- **Devices operating here:** Switches (read destination MAC, forward to correct port), Wi-Fi access points
- **Breaks when:** Duplicate MACs on segment, switch port misconfiguration, VLAN mismatch, ARP cache poisoning, half-duplex mismatch causing collisions
- **Diagnose:** `arp -a`, Wireshark Layer 2 capture, check switch port status, look for CRC errors on interface

Layer 3 — Network

- Logical addressing (IP), routing, fragmentation
- **Real-world:** IPv4/IPv6 packets, ICMP, routers
- **Devices operating here:** Routers, Layer 3 switches
- **Breaks when:** Wrong IP/subnet/gateway configured, routing table missing, NAT misconfiguration, firewall blocking
- **Diagnose:** `ping`, `tracert`/`traceroute`, `route print`, `ip route show`

Layer 4 — Transport

- End-to-end communication, multiplexing via ports, reliability
- TCP: connection-oriented, guaranteed delivery, ordered, flow-controlled
- UDP: connectionless, best-effort, no ordering guarantee, lower overhead
- **Real-world:** Port 80/443 for HTTP/HTTPS, port 22 for SSH, port 53 UDP for DNS

- **Breaks when:** Port blocked by firewall, wrong port number, SYN/ACK not returning (asymmetric routing), buffer overflow
- **Diagnose:** `netstat`, `ss`, `Test-NetConnection -Port`, `nmap`, Wireshark TCP stream analysis

Layer 5 — Session

- Manages sessions between applications: establishing, maintaining, terminating
- In modern TCP/IP, largely handled by the application and OS together
- **Real-world:** TLS session establishment, SMB session tokens, RPC sessions
- **Breaks when:** Session timeouts (keepalive misconfigured), authentication tokens expire, NAT state table entry expires
- **Diagnose:** Application logs, Wireshark session analysis

Layer 6 — Presentation

- Data encoding, encryption, compression
- **Real-world:** TLS/SSL encryption, JPEG/PNG compression, character encoding (UTF-8)
- **Breaks when:** SSL certificate mismatch, cipher suite incompatibility, encoding mismatch
- **Diagnose:** `openssl s_client -connect host:443`, browser cert errors

Layer 7 — Application

- The protocol the application itself speaks
- **Real-world:** HTTP, HTTPS, DNS, SMTP, FTP, SSH, SNMP
- **Breaks when:** Application misconfiguration, API endpoint wrong, DNS name wrong, server-side errors
- **Diagnose:** Application-specific tools (`curl`, `wget`, browser dev tools, `dig`)

Where Devices Operate

Device	OSI Layers	Notes
Repeater/Hub	L1	Dumb signal amplifier, no frame awareness
Switch (unmanaged)	L2	MAC table forwarding
Switch (managed/L3)	L2–L3	Can do inter-VLAN routing
Wi-Fi NIC	L1–L2	Physical radio + 802.11 frame handling
Router	L3	IP routing decisions
Firewall	L3–L7	Depends on type; stateful = L3-4, NGFW = L7
Load balancer	L4 or L7	L4 = TCP, L7 = HTTP/HTTPS aware

Layer-by-Layer Diagnostic Order

When something doesn't work, go bottom-up:

1. **L1:** Is the cable plugged in? Is the link light on? Is Wi-Fi associated? Check physical
2. **L2:** Is there an ARP entry? Can you see the gateway's MAC? No VLAN issue?
3. **L3:** Can you ping the gateway? Can you ping 8.8.8.8? Check routing table
4. **L4:** Can you reach the port? (`telnet 8.8.8.8 80` or `Test-NetConnection`)
5. **L5–7:** Is the application responding? Is TLS working? Is the protocol speaking correctly?

DNS — Domain Name System

What It Does and Why It Exists

DNS translates human-readable names (`google.com`) into IP addresses (`142.250.80.46`). Without it, you'd need to memorize IPs for every service.

The internet has no single authoritative list of all names. Instead, DNS is a **distributed, hierarchical, delegated database**. Each domain owner manages their own zone. The hierarchy:

```
. (root)
└── com. (TLD — Top Level Domain)
    └── google.com. (Second Level Domain, managed by Google)
        └── mail.google.com. (subdomain, still managed by Google)
```

Every domain name technically ends in a dot (the root), though you don't type it.

Resolution Chain: Recursive vs Iterative

When you type `google.com` into your browser:

Recursive resolution (what your device does):

1. Your device asks its configured **DNS resolver** (usually your router or ISP): "What is google.com's IP?"
2. The resolver does the work and returns a final answer

Iterative resolution (what the resolver does internally):

1. Resolver asks a **root nameserver** (there are 13 logical root servers, operated by various orgs): "Who handles .com?"
2. Root says: "I don't know, but ask one of the .com TLD nameservers: `192.5.6.30`"
3. Resolver asks TLD nameserver: "Who handles google.com?"
4. TLD says: "I don't know, but ask Google's nameservers: `ns1.google.com`"
5. Resolver asks `ns1.google.com`: "What is google.com's A record?"
6. Google's nameserver answers: `142.250.80.46`
7. Resolver caches the answer (for the TTL duration) and returns it to your device

Caching is critical for performance and is why DNS propagation takes time — old cached records persist until their TTL expires. TTL is set by the zone owner and can be anything from 60 seconds to 86400 seconds (1

day).

DNS Record Types

A Record — Maps a hostname to an IPv4 address

```
google.com.  300  IN  A  142.250.80.46
```

AAAA Record — Maps a hostname to an IPv6 address

```
google.com.  300  IN  AAAA  2607:f8b0:4004:c1b::64
```

CNAME (Canonical Name) — Maps a hostname to another hostname (alias)

```
www.example.com.  3600  IN  CNAME  example.com.
```

Important: CNAME cannot coexist with other records at the same name (you can't have a CNAME at the zone apex alongside an SOA). That's why "ALIAS" or "ANAME" records exist in some providers.

MX (Mail Exchanger) — Specifies mail servers for a domain, with priority

```
example.com.  3600  IN  MX  10  mail1.example.com.
example.com.  3600  IN  MX  20  mail2.example.com.
```

Lower priority number = higher priority. `10` is tried before `20`.

PTR (Pointer) — Reverse DNS: maps IP to hostname. Lives in `in-addr.arpa.` zone

```
46.80.250.142.in-addr.arpa.  300  IN  PTR  lga34s14-in-f14.1e100.net.
```

Used by mail servers to verify sender legitimacy, security tools, logging.

TXT (Text) — Arbitrary text. Heavily used for:

- SPF (Sender Policy Framework): defines which IPs can send email for a domain
- DKIM public keys
- Domain verification (Google, Microsoft, etc.)
- DMARC policies

```
example.com.  3600  IN  TXT  "v=spf1 include:_spf.google.com ~all"
```

NS (Name Server) — Delegates a zone to authoritative nameservers

```
example.com.  172800  IN  NS  ns1.example.com.
```

SOA (Start of Authority) — One per zone, contains zone metadata: primary NS, admin email, serial number, refresh/retry/expire/minimum TTL values.

Local DNS vs Upstream

Your home router typically runs a small DNS forwarder (e.g., dnsmasq). Your devices send DNS queries to the router (`192.168.1.1`), and the router forwards them upstream to either:

- Your ISP's DNS resolvers (default)
- Public resolvers like `8.8.8.8` (Google), `1.1.1.1` (Cloudflare), `9.9.9.9` (Quad9)

Local DNS also handles:

- **DHCP-assigned hostnames:** When a device gets a DHCP lease, the router may register its hostname in local DNS so `mydevice.local` resolves to its IP
- **Local overrides:** You can add custom DNS entries in your router's config

---

/etc/hosts and Windows hosts file

The hosts file is processed **before** DNS queries. It maps names directly to IPs at the OS level, bypassing any DNS server entirely.

Linux: `/etc/hosts`

Windows: `C:\Windows\System32\drivers\etc\hosts`

Format (same on both):

```
127.0.0.1    localhost
::1          localhost
192.168.1.50  myserver myserver.local
0.0.0.0      ads.doubleclick.net    # blocks this domain
```

Why it overrides: The OS's **resolver order** is defined in `/etc/nsswitch.conf` (Linux) or registry (Windows). By default: `files` (hosts file) comes before `dns`. So the OS checks hosts first; if found, stops there.

Practical uses:

- Override a domain for local testing (`myapp.com → 127.0.0.1` while developing)
- Block ad domains (point them to `0.0.0.0`)
- Force a specific IP for a name regardless of DNS
- Resolve hostnames without DNS server (isolated network)

> ***Note:*** `/etc/hosts` *changes take effect immediately — no restart or flush needed in most cases. However, browsers may cache DNS separately. In Chrome:* `chrome://net-internals/#dns` *to flush browser cache. In Windows:* `ipconfig /flushdns` *flushes the OS resolver cache (does not affect hosts file processing order).*

DHCP — Dynamic Host Configuration Protocol

The DORA Process

DORA is the four-step handshake between a client and DHCP server. It happens over UDP (client: 68, server: 67) and uses **broadcast** because the client doesn't know the server's IP yet.

Step 1 — DISCOVER

- Client broadcasts: "Is there a DHCP server? I need an IP address."
- Source: `0.0.0.0:68` → Destination: `255.255.255.255:67`
- Client includes its MAC address in the request

Step 2 — OFFER

- DHCP server responds: "I have `192.168.1.50` available. Take it for 24 hours."
- Server sends a unicast or broadcast offer to the client's MAC
- Offer includes: IP address, subnet mask, lease duration, default gateway, DNS servers

Step 3 — REQUEST

- Client broadcasts: "I'd like to accept the offer from server `192.168.1.1`."
- Still broadcast because multiple DHCP servers may have offered — this tells all servers which offer was chosen (others retract their offers)

Step 4 — ACK (Acknowledge)

- Server confirms: "IP `192.168.1.50` is yours for 24 hours."
- Client now configures its interface

If the requested address is unavailable (grabbed by another device during the process), the server sends a **NACK** (Negative Acknowledgment) and the process restarts.

Lease Times — What They Mean

The lease time defines how long the client may use the assigned IP. At 50% of the lease duration, the client begins trying to **renew** by sending a unicast REQUEST directly to the server that issued the lease (no broadcast needed this time — it knows the server's IP now).

If the server ACKs the renewal → lease extended. If the server doesn't respond (offline, etc.):

- At 87.5% of the lease time → client tries broadcast renewal (may reach a different DHCP server)
- At 100% (lease expires) → client must stop using the IP and restart DORA from scratch

What this means practically:

- Short leases (1–2 hours): DHCP pool recycles quickly; good for high-turnover environments (coffee shops, hotel Wi-Fi). More broadcast traffic.
- Long leases (24 hours–7 days): Devices tend to keep same IP. Better for stable environments. Reduces DHCP chatter.
- If your DHCP server goes offline while leases are active, clients will continue working until their lease expires. Then connectivity breaks.

DHCP Reservations (Static by MAC)

A DHCP reservation (sometimes called a "static DHCP lease" or "DHCP static mapping") tells the DHCP server: "When you see a DISCOVER from MAC `AA:BB:CC:DD:EE:FF`, always offer IP `192.168.1.100`."

The client still goes through DORA — it just always gets the same IP. This is different from a true static IP configured on the device itself.

Advantages over static IP on device:

- Central management — change the IP in one place (the router)
- The device can still fall back to DHCP if moved to a different network
- Works even on devices where you can't easily set a static IP

Disadvantage: If the DHCP server is unreachable during boot, the device gets nothing (or APIPA `169.254.x.x`). A truly static IP always configures regardless of any server.

Static IP vs DHCP — When to Use Each

Use Case	Recommendation
Servers, printers, cameras	Static IP (on device or via DHCP reservation)
IoT devices you need to reach consistently	DHCP reservation
Developer machines, always on the same network	Either; DHCP reservation is cleaner
Laptops that roam between networks	DHCP (auto-adapts)
Devices behind port forwarding rules	Static (or DHCP reservation) — NAT rules reference a specific IP
Emergency/isolated network	Static — no DHCP server available

Configuring a true static IP requires: IP address, subnet mask, default gateway, DNS server(s). If any of these are wrong or missing, specific things break in predictable ways:

- Wrong subnet mask → local/remote decision is wrong
- Wrong gateway → can reach local but not internet
- No DNS → can ping `8.8.8.8` but can't reach `google.com`

Gateways and Routing

What a Default Gateway Actually Does

The default gateway is the IP address of the router on your local segment — specifically, the router interface that's on the same subnet as your device.

When your device determines that a destination IP is **not local** (via the AND operation described earlier), it doesn't know how to reach it. It only knows one thing: "Send it to the gateway and let the gateway figure it out."

The device ARPs for the gateway's MAC, then sends the packet with:

- **Destination IP:** The actual target (e.g., `142.250.80.46`)
- **Destination MAC:** The gateway's MAC (the next-hop Layer 2 address)

The router receives the frame, strips the Layer 2 header, reads the IP, looks up its routing table, and forwards it toward the destination — potentially through many more routers.

Key insight: The IP packet's destination address never changes in transit (unless NAT is involved). Only the Layer 2 frame headers change at each hop.

How Routing Tables Work

Every device (including your laptop) has a routing table — a list of "for traffic going to network X, use next-hop Y via interface Z."

```
# Windows: route print
# Linux: ip route show
```

Typical routing table on a home machine:

Network	Mask	Gateway	Interface	Metric
0.0.0.0	0.0.0.0	192.168.1.1	eth0	100
192.168.1.0	255.255.255.0	On-link	eth0	10
127.0.0.0	255.0.0.0	On-link	lo	1

Match logic: The router uses **longest prefix match** — the most specific (most bits matched) route wins. `192.168.1.0/24` is more specific than `0.0.0.0/0`. So traffic for `192.168.1.50` matches the `/24` route, not the default route.

`0.0.0.0/0` is the **default route** — matches everything. If no more specific route exists, traffic goes here (to the gateway). That's the "default" in "default gateway."

Metric: When two routes match equally, the one with the lower metric wins. Metrics can represent hop count, bandwidth, reliability depending on the routing protocol.

Static Routes vs Dynamic Routing

Static routes are manually added entries in the routing table that don't change unless you change them:

```
# Linux: route traffic to 10.10.0.0/16 via 192.168.1.254
ip route add 10.10.0.0/16 via 192.168.1.254

# Windows:
route add 10.10.0.0 mask 255.255.0.0 192.168.1.254
```

Use static routes when:

- You have a VPN and need specific traffic to go through it
- You have two network interfaces and need traffic split between them
- Small, stable network where dynamic routing is overkill
- You're troubleshooting and need to force traffic a specific way

Dynamic routing protocols automatically discover routes and adapt to changes:

- **RIP (Routing Information Protocol):** Old, simple, counts hops, slow convergence. Still seen on old hardware.
- **OSPF (Open Shortest Path First):** Link-state, fast convergence, scales well, used in enterprise networks
- **BGP (Border Gateway Protocol):** The protocol that runs the internet. Exchanges routing information between autonomous systems (ISPs, big companies). Every major ISP runs BGP.
- **EIGRP:** Cisco proprietary, fast, hybrid distance-vector/link-state

For a networking technician: you'll encounter static routes and OSPF most often. BGP matters if you're in an ISP or large enterprise environment.

NAT — Why Your 192.168.x.x Gets to the Internet

NAT (Network Address Translation) is the mechanism that lets many private-IP devices share a single public IP.

How it works (PAT — Port Address Translation, the common case):

1. Your device (`192.168.1.50`) sends a packet to `142.250.80.46:443`
   - Source: `192.168.1.50:54231` (random ephemeral port)
   - Destination: `142.250.80.46:443`
2. The packet hits your router. The router:
   - Records in its **NAT table**: `192.168.1.50:54231 ↔ 203.0.113.5:54231` (your public IP + same or different port)
   - Rewrites the source IP+port to the public IP+port
   - Forwards the packet
3. Google's server responds to `203.0.113.5:54231`
4. The router receives the response, looks up its NAT table, rewrites the destination back to `192.168.1.50:54231`, and delivers it to your device

The port is the key. Without port tracking, the router couldn't tell which of 100 devices a returning packet belongs to. This is called **PAT (Port Address Translation)**, **IP Masquerading** on Linux, or just "NAT" colloquially.

Port forwarding is a static NAT rule: "Any incoming traffic on public port 8080, forward to private `192.168.1.100:80`." Without it, the NAT table has no unsolicited inbound entry, so external traffic can't reach internal services.

Hairpin NAT (NAT loopback): Allows a device inside your network to reach an internal server using the server's public IP/domain name. Not all routers support this properly.

PART 2: NETWORK ADAPTER SETTINGS

This section walks through every setting you'd see on an iPhone Wi-Fi detail screen, a Windows network adapter properties dialog, or Linux `ip`/`iw`/`nmcli` output. For each setting: what it is, what changing it does, when to change it, and how it interacts with other settings.

IP Address (Static vs DHCP)

What it is: The Layer 3 identifier assigned to the interface. Either assigned automatically by a DHCP server or configured manually (static).

DHCP mode: The device broadcasts DORA, receives IP/mask/gateway/DNS. The address may change between sessions (unless a DHCP reservation exists on the router).

Static mode: You manually specify IP, subnet mask, gateway, and DNS. The address never changes regardless of network state.

When to change:

- Set static if: this device needs to be reachable consistently (server, printer, IoT device behind a port forward rule)
- Keep DHCP if: device roams between networks, or you manage IPs centrally via reservations

Interactions:

- Setting a static IP that conflicts with another device → both devices will have issues; ARP will be confused
- Setting a static IP outside the subnet of the gateway → you can configure the address but traffic won't route properly
- Setting a static IP within the DHCP pool range → the DHCP server may later assign that IP to a different device (configure static IPs outside the DHCP pool range, or use reservations)

iPhone: Settings → Wi-Fi → tap (i) → Configure IP → Manual

Windows: Network adapter → Properties → IPv4 → Use the following IP address

Linux (ip):

```
ip addr add 192.168.1.50/24 dev eth0
ip route add default via 192.168.1.1
```

Subnet Mask

What it is: Defines which portion of the IP address is the network and which is the host. (See Part 1 for full explanation of ANDing.)

When to change: Only when the network design changes. If your admin tells you the network is `10.0.0.0/22`, set the mask to `255.255.252.0`. Mismatch causes subtle failures.

What misconfiguring does:

- Mask too narrow (e.g., `/25` on a `/24` network): device thinks some local hosts are remote → tries to route to them via gateway → gateway doesn't know about them → connection fails
- Mask too wide (e.g., `/16` on a `/24` network): device thinks remote hosts are local → tries to ARP for them → no response → connection fails

---

Router / Default Gateway

What it is: The IP address of the router interface on the local subnet. All non-local traffic goes here first.

When to change: Only when the router's IP changes or you're reconfiguring the network. Rarely needs manual intervention in DHCP environments (DHCP provides this).

What misconfiguring does:

- Wrong gateway IP → device can reach local hosts (Layer 2 works) but cannot access anything outside the subnet. `ping 8.8.8.8` fails, `ping 192.168.1.x` succeeds.
- No gateway → same result
- Gateway that doesn't actually route → device sends packets that disappear

---

DNS Servers (Primary, Secondary)

What it is: The IP addresses of DNS resolvers the device will query for name resolution.

Primary: Queried first. **Secondary:** Fallback if primary doesn't respond within timeout.

Common values:

- `8.8.8.8` / `8.8.4.4` — Google Public DNS
- `1.1.1.1` / `1.0.0.1` — Cloudflare (claims no-logging, slightly faster in many regions)
- `9.9.9.9` / `149.112.112.112` — Quad9 (security-focused, blocks malware domains)
- `192.168.1.1` — Your local router (common default; router forwards to upstream)

When to change:

- Your ISP's DNS is slow or hijacks NXDOMAIN (returns ads for nonexistent domains)
- You want filtering (Quad9 blocks malware, OpenDNS has parental controls)
- You're on a corporate network and need to resolve internal hostnames → use the corporate DNS
- Split DNS environments: internal DNS for internal names, external DNS for everything else

Interactions:

- If DNS is unreachable, applications fail with hostname-not-found even though IP connectivity is fine. This is one of the most common "internet is broken" scenarios that's actually a DNS issue.
- Using a public DNS on a corporate network may prevent resolution of internal names like `fileserver.corp.local`
- Some routers intercept DNS queries on port 53 and redirect them to themselves regardless of what you set (DNS hijacking for parental control features)

Search Domains

What it is: A list of domain suffixes appended to unqualified hostnames during DNS resolution.

Example: Search domain `corp.example.com`. You type `fileserver` → OS tries `fileserver.corp.example.com` first before trying bare `fileserver`.

When to change: When you're on a corporate or lab network where you frequently reference hosts by short name. VPN connections often push search domains so you can type `jira` and reach `jira.corp.company.com`.

Linux `/etc/resolv.conf`:

```
search corp.example.com example.com
```

Windows: Set per-adapter via Advanced TCP/IP settings → DNS → DNS suffix for this connection

Interactions: Multiple search domains are tried in order. Can cause unexpected delays if the first domain query times out before trying the next. Also a security note: attackers can potentially use search domain hijacking to intercept short-name queries.

HTTP Proxy (Manual, Auto, PAC Files)

What it is: A proxy intercepts your HTTP/HTTPS traffic and forwards it to the destination. The OS (and applications that respect OS proxy settings) route web requests through the proxy instead of directly.

Manual proxy: You specify host and port (e.g., `proxy.company.com:8080`). All HTTP/HTTPS goes through it.

Auto proxy / PAC file (Proxy Auto-Config): You specify a URL to a `.pac` file. The OS downloads and executes this JavaScript file for each connection to determine whether to use a proxy and which one. PAC files can route different domains through different proxies or no proxy.

PAC file example:

```
function FindProxyForURL(url, host) {
    if (isInNet(host, "10.0.0.0", "255.0.0.0")) {
        return "DIRECT";        // internal traffic goes direct
    }
    return "PROXY proxy.corp.com:8080";  // everything else through proxy
}
```

WPAD (Web Proxy Auto-Discovery): Protocol that allows automatic discovery of a PAC file via DHCP option 252 or DNS lookup of `wpad.yourdomain.com`. Means users get proxy settings automatically without manual configuration.

When to use:

- Corporate environments that filter/log all web traffic through a proxy

- Networks that require proxy authentication before allowing internet access
- Bypassing geographic restrictions (though VPNs are more common now)
- Transparent proxies: some networks intercept HTTP traffic at the network level without requiring OS proxy settings

Security note: WPAD has been abused in attacks (WPAD hijacking). Windows now requires DNS for WPAD; disabling WPAD auto-detection improves security.

Windows:

```
Settings → Network & Internet → Proxy
Or: Internet Options → Connections → LAN Settings
```

Linux (environment variable, used by curl/wget/etc.):

```
export http_proxy="http://proxy.corp.com:8080"
export https_proxy="http://proxy.corp.com:8080"
export no_proxy="localhost,127.0.0.1,10.0.0.0/8"
```

Private Wi-Fi Address (MAC Randomization)

What it is: Instead of using the burned-in hardware MAC address, the device uses a randomly generated MAC per network (iOS) or per session (some Android implementations).

Apple's implementation (iOS 14+/iPadOS 14+/watchOS 7+): Each SSID gets a stable random MAC (consistent across connections to that SSID, but different from the real MAC and different for each SSID). The MAC persists until you forget the network or manually rotate it.

Why it exists: Wi-Fi access points log MAC addresses. If you use your real MAC everywhere (coffee shops, airports, retail stores), your device's physical presence can be tracked across locations. Randomized MACs prevent this cross-network tracking.

When to disable:

- Network uses MAC-based access control (allowlist) and you need to add the real MAC
- DHCP reservation on your home router is tied to the device's real MAC
- Enterprise Wi-Fi with certificate + MAC binding
- Certain captive portals that identify you by MAC

iPhone: Settings → Wi-Fi → tap (i) → Private Wi-Fi Address → toggle off

Android: Wi-Fi settings → tap network → Privacy → Use device MAC

Linux (NetworkManager):

```
nmcli connection modify "MyNetwork" wifi.cloned-mac-address random
# or stable (generates consistent random per SSID):
nmcli connection modify "MyNetwork" wifi.cloned-mac-address stable
# or use real MAC:
nmcli connection modify "MyNetwork" wifi.cloned-mac-address permanent
```

Limit IP Address Tracking (iCloud Private Relay)

What it is: An Apple feature (requires iCloud+ subscription) that routes Safari traffic through two Apple/third-party relay servers. Your real IP is hidden from the destination website, and Apple's servers don't know what site you're visiting (split between ingress and egress nodes).

Not a full VPN: Only affects Safari and some DNS queries. Other apps still use your real IP unless they also route through relays.

When to disable: Corporate networks or enterprise MDM profiles may block it. Speeds may be slightly reduced. Not useful when you need your real IP visible (logging into region-specific content).

Low Data Mode

iPhone: Reduces background network activity — pauses iCloud sync, disables automatic updates, reduces video streaming quality. Useful on metered connections (cellular hotspots with data caps).

Windows 10/11: Set a connection as "Metered." Apps respect this differently; the OS stops downloading Windows updates automatically.

No effect on IP configuration or routing. It's an application-layer hint.

Auto-Join

What it is: Whether the device automatically connects to this Wi-Fi network when in range.

When to disable: You want to be in range of a network but only connect manually (e.g., a neighbor's SSID your device keeps joining, or a weak signal network you don't want preferentially chosen).

Interaction with network priority: iOS and Windows both maintain a network priority list. When multiple known SSIDs are in range, the highest-priority one with Auto-Join enabled is chosen. You can reorder the list.

Wi-Fi Channel and Band (2.4 vs 5 vs 6 GHz)

2.4 GHz band:

- Longer range (lower frequency penetrates walls better)

- Lower maximum throughput
- Only 3 non-overlapping channels in the US: 1, 6, 11 (channel spacing is 5 MHz, each channel is 22 MHz wide)
- Heavily congested in dense environments (every neighbor's router, Bluetooth, microwave ovens, baby monitors, ZigBee)
- Maximum theoretical: 802.11n = 600 Mbps, 802.11g = 54 Mbps in practice

5 GHz band:

- Shorter range (higher frequency attenuates more in walls)
- Much higher throughput
- Many more non-overlapping channels (23 in the US across UNII-1, UNII-2A, UNII-2C, UNII-3 bands)
- Less congested
- DFS channels (channels 52–144): require Dynamic Frequency Selection to avoid radar interference. Access points must scan for radar before using; this causes brief connection drops

6 GHz band (Wi-Fi 6E, 802.11ax):

- 1200 MHz of spectrum vs 500 MHz for 5 GHz
- Up to 59 non-overlapping 20 MHz channels
- No legacy devices allowed — cleaner spectrum
- Shorter range than 5 GHz due to higher frequency
- Requires Wi-Fi 6E hardware on both AP and client

Which to choose: For stationary devices that need speed and the AP is nearby → 5 GHz. For range, IoT devices, legacy hardware → 2.4 GHz. For maximum throughput on Wi-Fi 6E hardware → 6 GHz.

Channel selection interaction: All APs on the same channel within range (even neighbor's APs) compete for airtime in CSMA/CA. You want your AP on a channel nobody nearby is using. On 2.4 GHz: use only 1, 6, or 11. On 5 GHz: most channels are non-overlapping at 20 MHz width.

Channel Width (20 / 40 / 80 / 160 MHz)

What it is: How much spectrum the AP and client use for a single channel. Wider = more throughput (more data lanes), but uses more spectrum and thus creates more interference.

Width	Bands	Max Streams (802.11ac)	Notes
20 MHz	2.4, 5, 6	Baseline	Safest in congested environments
40 MHz	2.4, 5, 6	2×	On 2.4 GHz, almost impossible without overlap
80 MHz	5, 6	4×	Common for modern 5 GHz APs

Width	Bands	Max Streams (802.11ac)	Notes
160 MHz	5, 6	8×	Maximum throughput; consumes huge 5 GHz chunk

Interaction with congestion: Wider channels bond adjacent channels. If neighbors are on those channels, you cause and receive interference. In dense deployments (apartments), 20 MHz on 5 GHz often outperforms 80 MHz in real throughput.

Auto channel width: Most modern APs have "auto" which can dynamically narrow the channel if radar or high interference is detected.

Transmit Power

What it is: How much RF power the radio outputs. Measured in dBm or mW (1 mW = 0 dBm, 100 mW = 20 dBm; decibels are logarithmic so +3 dB ≈ doubles power, +10 dB = 10× power).

Regulatory limits: Each country has maximum transmit power limits (FCC in US, ETSI in Europe). The `iw reg` command shows your regulatory domain settings.

Why more isn't always better:

- Your device (phone/laptop) has limited transmit power (~15–20 dBm typical). If the AP blasts at 30 dBm, the AP can hear itself fine, but the client's weaker signal may not make it back — "deaf AP" problem.
- High transmit power increases interference range — your signal stomps on neighbors' channels
- Optimal: AP and client transmit power matched; AP power reduced until SNR is good without creating long interference range

Linux:

```
iw dev wlan0 set txpower fixed 1500   # 15 dBm (value in mBm = 100 * dBm)
iw phy phy0 set txpower auto
```

RTS/CTS Threshold

What it is: RTS/CTS (Request To Send / Clear To Send) is a mechanism to address the **hidden node problem** in 802.11.

Hidden node problem: Device A and Device C can both hear the AP but cannot hear each other. A and C both transmit simultaneously → collision at the AP → both frames corrupted.

RTS/CTS solution: Before transmitting a large frame, the sender transmits a small RTS frame. The AP responds with CTS. All devices that hear the CTS go silent for the specified duration. This prevents collisions from hidden nodes.

Threshold: The minimum frame size (in bytes) that triggers RTS/CTS. Default is typically 2347 (effectively disabled, since max frame is ~2346 bytes). Setting to 500 means any frame over 500 bytes uses RTS/CTS.

When to change:

- Environments with many hidden nodes (large open spaces, directional antennas)
- High collision rate → enable RTS/CTS by lowering threshold
- Adds overhead, so lower threshold = more overhead. Only useful when collisions are a real problem.

Fragmentation Threshold

What it is: Maximum size (bytes) of a Wi-Fi frame before it's split into fragments. Default is typically 2346 (effectively disabled). Range: 256–2346 bytes.

Why it exists: In high-interference environments, smaller frames are less likely to be corrupted during transmission. A corrupted fragment only requires retransmission of that fragment, not the whole large frame.

When to use: Rarely in modern environments. Useful in environments with high RF interference where large frames get corrupted frequently. Setting too low creates overhead that worsens overall throughput. The standard recommendation is to leave it at default.

Beacon Interval

What it is: How often the AP broadcasts beacon frames (in Time Units, 1 TU = 1.024 ms). Default is typically 100 TU (≈102.4 ms → roughly 10 beacons/second).

Beacon frames announce:

- SSID (unless hidden)
- Supported rates
- Channel info
- Security capabilities
- DTIM count
- QoS info
- Timestamp for synchronization

When to change:

- Increase interval (e.g., 200–300 TU): reduces overhead for fixed/wired-like stations; saves battery on battery-powered APs (some embedded systems)
- Decrease interval: not typically useful; increases overhead

Battery impact: Clients use beacons to wake up and check for buffered data (via DTIM). Longer beacon intervals = clients sleep longer = more battery savings but higher latency for buffered data.

DTIM Period

What it is: DTIM (Delivery Traffic Indication Message) is a beacon that tells sleeping clients: "I have buffered data for you, wake up." The DTIM period defines how many regular beacons occur between DTIM beacons.

- DTIM period 1: Every beacon is a DTIM. Clients check for buffered data every beacon interval (≈100ms). Low latency, less battery savings.
- DTIM period 2: Every other beacon is DTIM. Clients sleep 2 × beacon interval between checks.
- DTIM period 3: Every 3rd beacon. More sleep time.

Interaction: DTIM × beacon interval = sleep cycle. Default is often DTIM 1 or DTIM 2.

When to change:

- Battery-sensitive IoT devices on Wi-Fi: increase DTIM → more sleep = longer battery life, at the cost of multicast/broadcast delivery latency
- VoIP or real-time apps: keep DTIM 1 or 2 so buffered packets aren't delayed

---

WMM (Wi-Fi Multimedia / QoS)

What it is: WMM (Wi-Fi Multimedia) is the 802.11e quality-of-service implementation. It defines four access categories with different priorities and contention parameters:

Access Category	Priority	AIFSN	CWmin	CWmax	Typical Use
AC_BK (Background)	Lowest	7	15	1023	File downloads, bulk transfers
AC_BE (Best Effort)	Normal	3	15	1023	General web traffic
AC_VI (Video)	High	2	7	15	Video streaming
AC_VO (Voice)	Highest	2	3	7	VoIP, real-time audio

Higher priority classes have shorter waiting times (AIFSN = Arbitration Interframe Space Number) and smaller contention windows (CW), so they get access to the medium faster.

Required for 802.11n and later: WMM is mandatory for 802.11n certification. Disabling WMM on an 802.11n/ac/ax network disables MIMO power save and can actually reduce performance.

When to disable: Almost never. Some very old devices (pre-2004) don't support WMM and may have issues; in that case, disabling can stabilize the old device at the cost of QoS for all others.

802.11 Protocol Mode (b/g/n/ac/ax)

The full lineage:

Standard	Year	Bands	Max Theoretical	Key Tech
802.11a	1999	5 GHz	54 Mbps	OFDM
802.11b	1999	2.4 GHz	11 Mbps	DSSS
802.11g	2003	2.4 GHz	54 Mbps	OFDM
802.11n (Wi-Fi 4)	2009	2.4/5 GHz	600 Mbps	MIMO, 40 MHz channels
802.11ac (Wi-Fi 5)	2013	5 GHz	6.9 Gbps	MU-MIMO, 80/160 MHz, 256-QAM
802.11ax (Wi-Fi 6/6E)	2019	2.4/5/6 GHz	9.6 Gbps	OFDMA, TWT, BSS coloring

Mode setting on AP: Controls which modes are allowed. Setting to "N only" or "AC only" kicks off legacy devices. "Mixed" mode allows all but degrades efficiency — legacy frames slow the whole network (802.11b clients cause the AP to transmit protection frames for every packet, dramatically reducing throughput for everyone).

OFDMA (Wi-Fi 6): Allows a single transmission to simultaneously serve multiple clients in different frequency sub-bands. Major throughput improvement in dense environments.

TWT (Target Wake Time, Wi-Fi 6): AP and client negotiate when the client will be awake. Dramatically improves battery life for IoT devices.

BSS Coloring (Wi-Fi 6): Reduces inter-BSS interference by color-coding frames, allowing devices to ignore transmissions from other networks on the same channel.

BSSID vs SSID

SSID (Service Set Identifier): The network name — what users see. A string up to 32 bytes. Multiple APs can share the same SSID (e.g., enterprise campus Wi-Fi with dozens of APs all broadcasting "Corp-WiFi").

BSSID (Basic Service Set Identifier): The MAC address of the specific access point radio. Unique per AP per radio (a dual-band AP has two BSSIDs, one per band). This is the actual Layer 2 identity of the AP.

Why the distinction matters:

- When roaming across a campus, your device connects to the SSID but associates with specific BSSIDs. You can watch it roam by tracking BSSID changes.
- Rogue AP attacks often clone an SSID but can't clone the BSSID without also being on the same physical hardware.

- `netsh wlan show networks mode=bssid` (Windows) and `iw dev wlan0 scan` (Linux) show BSSIDs.
- In monitor mode packet captures, you see BSSIDs, not just SSIDs.

Hidden Networks

What it is: An AP with beacon suppression — the AP doesn't include the SSID in beacon frames. The network exists but doesn't appear in scan results.

Why it's weak security: The SSID is still transmitted in:

- Probe Request frames from clients actively searching for it
- Probe Response frames from the AP when a client asks
- Association Request/Response frames when connecting

A passive Wireshark capture in monitor mode reveals the SSID within seconds of any client connecting.

What it does affect:

- Clients that don't have the network configured can't see it (casual passerby won't connect)
- Can cause connectivity issues if the client's probe requests time out

Downsides:

- Windows/iOS clients that have a hidden network configured will actively probe for it by name wherever they are — this leaks your hidden SSID to every network you visit

Bottom line: Hidden SSIDs provide minimal security and create connectivity complexity. Use strong WPA3 + a long passphrase instead.

WPA2 / WPA3 Authentication Modes

WEP: Broken cryptographically. Never use. IVs too short, RC4 misused. Can be cracked in minutes.

WPA (WPA1): Intermediate fix using TKIP. Also broken in various ways. Avoid.

WPA2-Personal (PSK): Most common. Uses a Pre-Shared Key (passphrase) with CCMP/AES encryption. Vulnerable to offline dictionary attacks on the 4-way handshake (captured passively). Still secure with a strong, random passphrase.

WPA2-Enterprise: Each user/device has unique credentials (username+password or certificate) via 802.1X/RADIUS. Much stronger — compromise of one credential doesn't expose the whole network. The AP forwards authentication to a RADIUS server.

WPA3-Personal (SAE — Simultaneous Authentication of Equals): Replaces PSK with Dragonfly handshake. Key advantage: **forward secrecy** — even if the passphrase is captured later, past traffic can't be decrypted. Also resistant to offline dictionary attacks (each attempt requires live interaction with the AP).

WPA3-Enterprise: Adds optional 192-bit security mode, uses Suite B cryptography (CNSA algorithms). Targeted at government/high-security environments.

WPA3 Transition Mode: AP advertises both WPA2 and WPA3. Newer clients use WPA3, legacy clients use WPA2. This is the practical setting for most deployments transitioning to WPA3.

PMF — Protected Management Frames (802.11w)

What it is: Standard management frames (deauthentication, disassociation) in 802.11 are unencrypted and unauthenticated by default. An attacker can forge a deauth frame to any client, forcing disconnection (used in deauth attacks for WPA handshake capture).

PMF (802.11w) encrypts management frames, making forged deauth/disassoc frames detectable or rejected.

Modes:

- **Optional (PMF=capable):** Clients that support PMF use it; others connect without. No protection for legacy clients.
- **Required (PMF=required):** Only clients supporting PMF can associate. Breaks legacy clients. Required for WPA3.

WPA3 mandates PMF. WPA2 with PMF required is almost as good for management frame protection.

Why it matters: Without PMF, deauth attacks are trivial. With PMF required, deauth attacks are prevented for connected clients (though pre-association deauths can still occur in some implementations).

TKIP vs CCMP/AES

TKIP (Temporal Key Integrity Protocol):

- Designed as a quick fix for WEP, could be deployed via firmware update
- Still uses RC4 cipher under the hood, but with per-packet keys, sequence numbers, MIC
- Has known vulnerabilities (Beck-Tews, Ohigashi-Morii attacks)
- Capped at 54 Mbps — 802.11n requires AES for its higher rates
- Do not use TKIP. It is deprecated in WPA3 and should be avoided in WPA2.

CCMP (Counter Mode with CBC-MAC Protocol):

- Uses AES-128 in Counter mode + CBC-MAC for authentication
- Mandatory for 802.11n/ac/ax high-throughput modes
- No known practical attacks against CCMP itself
- "WPA2-AES" in consumer router UI typically means CCMP
- Always use CCMP over TKIP

GCMP (Galois/Counter Mode Protocol):

- Used in WPA3 (both 128-bit and 256-bit variants)
- More efficient than CCMP, slightly faster on hardware without dedicated AES acceleration
- Not commonly seen in consumer UI — it's used automatically with WPA3

Setting "WPA2-AES" vs "WPA2-TKIP+AES (Mixed)": Always choose AES-only unless you have legacy devices that don't support AES (very old, pre-2005 hardware). Mixed mode opens the door to TKIP vulnerabilities and limits throughput.

PART 3: WINDOWS NETWORKING — COMPLETE COMMAND REFERENCE

ipconfig

The fundamental Windows network diagnostic tool. Run from `cmd` (Command Prompt) — not PowerShell-specific, though it works there too.

`ipconfig`

```
ipconfig
```

Shows basic info for all interfaces: IP address, subnet mask, default gateway. If you see `169.254.x.x` (APIPA), the adapter got no DHCP response.

`ipconfig /all`

```
ipconfig /all
```

The full picture. Shows for each adapter:

- **Connection-specific DNS Suffix:** The DNS search suffix for this adapter (e.g., `corp.example.com`)
- **Description:** Human-readable adapter name (important when using netsh)
- **Physical Address:** The MAC address (or the randomized one if spoofed)
- **DHCP Enabled:** Yes/No
- **Autoconfiguration Enabled:** Whether APIPA fallback is active
- **IPv4 Address:** Followed by `(Preferred)` — means it was successfully configured
- **Lease Obtained / Lease Expires:** DHCP lease timestamps
- **DHCP Server:** IP of the DHCP server that issued the lease
- **DNS Servers:** Currently active DNS resolvers
- **NetBIOS over Tcpip:** Whether legacy Windows name resolution is enabled

`ipconfig /release`

```
ipconfig /release
ipconfig /release "Wi-Fi"    # release only a specific adapter
```

Sends a DHCP Release message to the server, relinquishing the IP. The adapter IP is set to `0.0.0.0`. Use before `/renew` to force a fresh lease cycle. Also useful when switching networks.

`ipconfig /renew`

```
ipconfig /renew
ipconfig /renew "Ethernet"   # renew only a specific adapter
```

Initiates a fresh DHCP request. If already leased, sends a DHCP Request to renew; if no current lease (after /release), performs full DORA. The adapter name in quotes must match the connection name exactly as shown in `ipconfig /all`.

`ipconfig /flushdns`

```
ipconfig /flushdns
```

Clears the Windows DNS resolver cache. The OS caches DNS responses (positive and negative) to speed up lookups. If you just changed a DNS record (or a /etc/hosts entry), you need this to force re-resolution. Does not flush the hosts file — hosts file changes take effect immediately.

When you need this:

- Changed a DNS record and the old IP is still being used
- A site that was broken is now fixed
- `/etc/hosts` changes not reflecting (though usually immediate)
- "It works on my phone but not my computer" — stale cache on the computer

`ipconfig /displaydns`

```
ipconfig /displaydns
ipconfig /displaydns | more    # paginate output
```

Shows the current contents of the DNS resolver cache. For each entry:

- Record name (hostname)
- Record type (A, CNAME, etc.)
- TTL remaining
- Data (the IP address or CNAME target)

Useful for verifying what IP a hostname is currently resolving to before flushing.

`ipconfig /registerdns`

```
ipconfig /registerdns
```

Forces the DNS client to re-register the machine's hostname and IP with the DNS server (Dynamic DNS update via RFC 2136). Used in domain environments where Windows machines dynamically register themselves in Active Directory DNS. Useful after changing a computer's IP so DNS updates immediately instead of waiting for the registration interval.

---

netsh

`netsh` (Network Shell) is a command-line scripting utility that provides deep access to network configuration. More powerful than `ipconfig`.

Show current IP configuration

```
netsh interface ip show config
netsh interface ip show config name="Ethernet"
```

Shows IP addresses, gateway, DNS, and WINS settings per interface.

Set a static IP address

```
netsh interface ip set address name="Ethernet" static 192.168.1.50 255.255.255.0 192.168.1.1
```

- `name=` must match the interface name exactly (get it from `ipconfig /all` or `netsh interface show interface`)
- Parameters: `static [IP] [mask] [gateway]`

Set back to DHCP

```
netsh interface ip set address name="Ethernet" dhcp
```

Set DNS servers

```
netsh interface ip set dns name="Ethernet" static 8.8.8.8
netsh interface ip add dns name="Ethernet" 8.8.4.4 index=2
```

First command sets primary DNS. Second adds secondary at index 2. Index=1 is primary.

Set DNS to automatic (DHCP)

```
netsh interface ip set dns name="Ethernet" dhcp
```

Show saved Wi-Fi profiles

```
netsh wlan show profiles
```

Lists all Wi-Fi profiles (SSIDs ever connected to, with saved credentials) on the machine.

Show Wi-Fi profile details including password

```
netsh wlan show profile name="MyNetwork" key=clear
```

The `key=clear` flag reveals the saved password in plaintext under "Key Content." This works for any stored Wi-Fi profile. **Security note:** Any local admin can do this. Stored Wi-Fi credentials on a shared machine are accessible.

Connect / disconnect Wi-Fi

```
netsh wlan connect name="MyNetwork"
netsh wlan connect name="MyNetwork" ssid="ActualSSID"    # if profile name differs from SSID
netsh wlan disconnect interface="Wi-Fi"
```

Show available networks with BSSID info

```
netsh wlan show networks mode=bssid
```

Shows nearby Wi-Fi networks including:

- SSID
- Network type (Infrastructure/Ad Hoc)
- Authentication method
- Encryption
- BSSID (AP MAC address)
- Signal strength (as percentage, not dBm — frustratingly)
- Channel
- Band

Show wireless interface details

```
netsh wlan show interfaces
```

Comprehensive info on the connected Wi-Fi interface:

- State (connected/disconnected)
- SSID and BSSID
- Network type
- Radio type (802.11n, ac, etc.)
- Authentication and cipher
- Channel
- Receive rate / Transmit rate (Mbps)
- Signal (%)
- Profile name

Set up a hosted network (Windows 7/8/early 10)

```
netsh wlan set hostednetwork mode=allow ssid="MyHotspot" key="password123"
netsh wlan start hostednetwork
netsh wlan stop hostednetwork
netsh wlan show hostednetwork
```

> ***Note:*** *The hosted network feature is deprecated in newer Windows 10/11 in favor of Mobile Hotspot. The* `netsh wlan set hostednetwork` *command may fail on newer builds. Use Settings → Network & Internet → Mobile Hotspot instead.*

Firewall commands

```
# Show all firewall rules
netsh advfirewall firewall show rule name=all

# Add inbound allow rule for a port
netsh advfirewall firewall add rule name="Allow Port 8080" dir=in action=allow protocol=TCP localport=8

# Add inbound block rule
netsh advfirewall firewall add rule name="Block Port 23" dir=in action=block protocol=TCP localport=23

# Delete a rule
netsh advfirewall firewall delete rule name="Allow Port 8080"

# Show firewall status
netsh advfirewall show allprofiles

# Turn off firewall (all profiles — don't leave this off)
netsh advfirewall set allprofiles state off

# Turn on firewall
netsh advfirewall set allprofiles state on

# Export/import firewall policy
netsh advfirewall export "C:\fw_backup.wfw"
netsh advfirewall import "C:\fw_backup.wfw"
```

PowerShell Networking

PowerShell provides richer network commands with structured object output. Run as Administrator for configuration changes.

Get-NetAdapter

```
Get-NetAdapter
Get-NetAdapter -Name "Ethernet"
Get-NetAdapter | Where-Object {$_.Status -eq "Up"}
Get-NetAdapter | Format-List *    # show all properties
```

Shows all network adapters with:

- Name, InterfaceDescription

- MacAddress
- Status (Up/Down/Disconnected)
- LinkSpeed
- MediaConnectionState

Get-NetIPAddress

```
Get-NetIPAddress
Get-NetIPAddress -InterfaceAlias "Ethernet"
Get-NetIPAddress -AddressFamily IPv4
Get-NetIPAddress | Where-Object {$_.IPAddress -notlike "169.*"}    # filter APIPA
```

Shows IP addresses for all interfaces. Key fields:

- IPAddress
- PrefixLength (CIDR notation — 24 means /24)
- InterfaceAlias
- AddressFamily (IPv4/IPv6)
- Type (Unicast/Multicast)
- AddressState (Preferred/Tentative/Invalid)

Get-NetIPConfiguration

```
Get-NetIPConfiguration
Get-NetIPConfiguration -InterfaceAlias "Wi-Fi"
Get-NetIPConfiguration -Detailed
```

A combined view: shows IP, subnet, gateway, and DNS for each interface. The `-Detailed` flag adds DHCP info, DNS server list, etc. Good first diagnostic command.

New-NetIPAddress (Set Static IP)

```
# First, get the interface index
Get-NetAdapter

# Set static IP
New-NetIPAddress -InterfaceAlias "Ethernet" -IPAddress 192.168.1.50 -PrefixLength 24 -DefaultGateway 19
```

If an IP already exists on the interface, this will error. Remove the existing one first:

Remove-NetIPAddress

```
Remove-NetIPAddress -InterfaceAlias "Ethernet" -IPAddress 192.168.1.50 -Confirm:$false
```

Use `-Confirm:$false` to suppress the interactive prompt.

Full static IP workflow:

```
# Remove existing DHCP-assigned IPs
Remove-NetIPAddress -InterfaceAlias "Ethernet" -Confirm:$false

# Remove existing gateway
Remove-NetRoute -InterfaceAlias "Ethernet" -DestinationPrefix "0.0.0.0/0" -Confirm:$false

# Set new static IP
New-NetIPAddress -InterfaceAlias "Ethernet" -IPAddress 192.168.1.50 -PrefixLength 24 -DefaultGateway 19

# Set DNS
Set-DnsClientServerAddress -InterfaceAlias "Ethernet" -ServerAddresses ("8.8.8.8","8.8.4.4")
```

Set-DnsClientServerAddress

```
Set-DnsClientServerAddress -InterfaceAlias "Ethernet" -ServerAddresses ("8.8.8.8","1.1.1.1")

# Reset to DHCP-provided DNS
Set-DnsClientServerAddress -InterfaceAlias "Ethernet" -ResetServerAddresses
```

Get-DnsClientServerAddress

```
Get-DnsClientServerAddress
Get-DnsClientServerAddress -InterfaceAlias "Wi-Fi"
Get-DnsClientServerAddress -AddressFamily IPv4
```

Shows configured DNS servers per interface.

Test-NetConnection

```
# Basic ping test
Test-NetConnection 8.8.8.8

# Test a specific TCP port
Test-NetConnection -ComputerName google.com -Port 443

# Traceroute
Test-NetConnection 8.8.8.8 -TraceRoute

# Diagnose mode (more detail)
Test-NetConnection -ComputerName google.com -DiagnoseRouting
```

This command does in one command what previously required `ping` + `tracert` + `telnet`/`portqry`. Output includes:

- PingSucceeded (True/False)
- PingReplyDetails (RTT, etc.)
- TcpTestSucceeded (True/False — for port tests)
- TraceRoute hop list
- RemoteAddress (resolved IP)

Get-NetRoute

```
Get-NetRoute
Get-NetRoute -AddressFamily IPv4
Get-NetRoute | Where-Object {$_.NextHop -ne "0.0.0.0"}    # non-local routes only
```

Shows the routing table. Key fields:

- DestinationPrefix (network/CIDR)
- NextHop (gateway IP)
- InterfaceAlias
- RouteMetric
- ifIndex (interface index number)

New-NetRoute

```
# Add a static route: traffic to 10.10.0.0/16 goes via 192.168.1.254
New-NetRoute -DestinationPrefix "10.10.0.0/16" -NextHop "192.168.1.254" -InterfaceAlias "Ethernet"

# Add with metric
New-NetRoute -DestinationPrefix "10.10.0.0/16" -NextHop "192.168.1.254" -InterfaceAlias "Ethernet" -Rou
```

Resolve-DnsName

```
# Basic lookup
Resolve-DnsName google.com

# Specific record type
Resolve-DnsName google.com -Type MX
Resolve-DnsName google.com -Type AAAA
Resolve-DnsName google.com -Type TXT

# Query specific DNS server
Resolve-DnsName google.com -Server 8.8.8.8

# Reverse lookup
Resolve-DnsName 142.250.80.46
```

More flexible than `nslookup`. Output is PowerShell objects that can be piped.

Get-NetAdapterStatistics

```
Get-NetAdapterStatistics
Get-NetAdapterStatistics -Name "Ethernet"
```

Shows per-adapter packet counters:

- ReceivedBytes, SentBytes
- ReceivedUnicastPackets, SentUnicastPackets
- ReceivedDiscardedPackets, SentDiscardedPackets

- ReceivedErrors, OutboundErrors

Errors and discarded packets indicate hardware issues, driver problems, or link problems.

---

ping

```
ping google.com
ping 8.8.8.8
ping 192.168.1.1 -t             # continuous ping until Ctrl+C
ping 192.168.1.1 -n 10          # send 10 packets
ping 192.168.1.1 -l 1400        # set packet size to 1400 bytes (test MTU issues)
ping 192.168.1.1 -f             # don't fragment (combine with -l to find MTU)
ping 192.168.1.1 -i 5           # set TTL to 5
ping ::1                         # ping IPv6 loopback
ping -6 google.com              # force IPv6
ping -4 google.com              # force IPv4
ping 192.168.1.1 -a             # resolve hostname from IP (reverse DNS)
```

Reading output:

```
Reply from 8.8.8.8: bytes=32 time=12ms TTL=118
```

- `bytes=32`: Size of the payload (default). Increase with `-l` to test fragmentation.
- `time=12ms`: Round-trip time. >150ms feels laggy for interactive traffic.
- `TTL=118`: Time-to-live of the returned packet. Linux default outgoing TTL is 64, Windows is 128, Cisco routers are 255. `TTL=118` suggests ≈10 hops from a Windows/Linux source (128-10=118 or 64-something).

"Request timed out" vs **"Destination host unreachable"**:

- Request timed out: Packet left your host but no ICMP echo reply returned. Could be firewall, host down, routing asymmetry.
- Destination host unreachable: Your local machine or a router on the path knows there's no route to the destination and sent back an ICMP unreachable message.

Ping with large payload to test MTU:

```
ping 192.168.1.1 -l 1472 -f
```

1472 + 28 (IP+ICMP headers) = 1500 bytes, the standard Ethernet MTU. If this fails with "Packet needs to be fragmented but DF set," your MTU is less than 1500.

---

tracert

```
tracert google.com
tracert -d 8.8.8.8          # don't resolve IP to hostname (faster)
tracert -h 20 google.com    # max 20 hops (default 30)
tracert -w 1000 google.com  # wait 1000ms per reply (default 4000ms)
tracert -6 google.com       # force IPv6
tracert -4 google.com       # force IPv4
```

How it works: tracert sends ICMP Echo Request packets with increasing TTL values (1, 2, 3...). When TTL expires at a router, that router sends back an ICMP "Time Exceeded" message, revealing its IP. Repeat until destination reached.

Reading output:

```
  1     1 ms     1 ms     1 ms  192.168.1.1
  2    12 ms    11 ms    13 ms  10.0.0.1
  3     *        *        *     Request timed out.
  4    25 ms    24 ms    26 ms  72.14.215.165
```

- Three RTT values per hop (three probes sent)
- `*` = that router didn't respond to ICMP (firewall blocking, router configured to not send TTL-exceeded for ICMP). Doesn't mean traffic stops — just that router is silent.
- A hop where RTT suddenly jumps indicates latency at that point (high-RTT hop, or congestion)
- If all remaining hops show `*` after a certain point, traffic is blocked or lost there

---

pathping

```
pathping google.com
pathping -n 8.8.8.8     # no hostname resolution
pathping -h 20 8.8.8.8  # max hops
pathping -p 100 8.8.8.8 # 100ms between pings (default 250ms)
pathping -q 50 8.8.8.8  # 50 pings per hop (default 100)
```

pathping combines tracert and ping. It first traces the route, then sends 100 pings to each hop and reports both per-hop RTT and **packet loss**. This reveals whether packet loss is occurring at a specific router in the path — information tracert doesn't give you.

Output shows two loss columns:

- `This Node/Link`: Loss at this router itself
- `Loss`: Loss on the link to the next hop

A hop showing 50% packet loss is often just QoS deprioritizing ICMP; if the final destination shows no loss, the intermediate loss doesn't matter.

---

nslookup

```
nslookup google.com                     # basic A record lookup
nslookup google.com 8.8.8.8            # query specific DNS server
nslookup -type=MX google.com           # MX records
nslookup -type=AAAA google.com         # IPv6 records
nslookup -type=TXT google.com          # TXT records
nslookup -type=NS google.com           # nameserver records
nslookup -type=PTR 46.80.250.142.in-addr.arpa.   # reverse lookup (the hard way)
nslookup -type=ANY google.com          # all records (often filtered by servers)
nslookup -debug google.com             # verbose debug output
```

Interactive mode:

```
nslookup
> server 8.8.8.8           # change server
> set type=MX
> google.com
> exit
```

Reading output:

```
Server:  dns.google
Address:  8.8.8.8

Non-authoritative answer:
Name:    google.com
Address:  142.250.80.46
```

"Non-authoritative" means the answer came from a cache, not directly from the authoritative nameserver. An authoritative answer comes directly from the domain's own nameservers and doesn't show this caveat.

---

netstat

```
netstat -a          # all connections and listening ports
netstat -n          # numeric (don't resolve hostnames) — faster
netstat -o          # show PID (process ID) for each connection
netstat -b          # show executable name (requires admin)
netstat -p TCP      # filter by protocol (TCP or UDP)
netstat -r          # show routing table (same as route print)
netstat -s          # per-protocol statistics (packets sent, errors, etc.)
netstat -e          # Ethernet statistics (bytes/packets sent/received)
netstat -an         # common combo: all + numeric
netstat -ano        # all + numeric + PID — the most useful
```

Reading `netstat -ano` output:

```
Proto  Local Address      Foreign Address    State        PID
TCP    0.0.0.0:80         0.0.0.0:0          LISTENING    4
TCP    192.168.1.50:54231 142.250.80.46:443  ESTABLISHED  1234
```

- `Local Address`: `0.0.0.0:80` means listening on all interfaces on port 80

- `127.0.0.1:8080` means listening only on loopback (local processes only)
- `State LISTENING`: Server socket waiting for connections
- `State ESTABLISHED`: Active connection
- `State TIME_WAIT`: Connection closed but waiting for delayed packets (normal, lasts ~60-120s)
- `State CLOSE_WAIT`: Remote side closed, waiting for local app to close its end. Large numbers suggest application not closing sockets.

To identify what's on port 80:

```
netstat -ano | findstr :80
tasklist | findstr "4"    # look up PID 4 (System)
```

---

arp

```
arp -a                    # show ARP cache for all interfaces
arp -a -N 192.168.1.50   # show ARP cache for specific interface IP
arp -d 192.168.1.1       # delete specific entry
arp -d *                  # delete all entries (requires admin)
arp -s 192.168.1.1 AA-BB-CC-DD-EE-FF  # add static ARP entry
```

Reading `arp -a` output:

```
Interface: 192.168.1.50 --- 0x4
  Internet Address    Physical Address      Type
  192.168.1.1         aa-bb-cc-dd-ee-ff     dynamic
  192.168.1.255       ff-ff-ff-ff-ff-ff     static
  224.0.0.22          01-00-5e-00-00-16     static
```

- `dynamic`: Learned via ARP, will expire
- `static`: Permanent entry (manually added or multicast/broadcast)
- `224.0.0.22` is a multicast group address; its MAC is derived mathematically

When to use:

- Verify a device is visible at Layer 2 (`ping` first to populate the cache, then check ARP)
- Troubleshoot ARP poisoning (unexpected MAC for a known IP)
- Find the MAC address of any device on the local subnet
- `/d` to force re-resolution after a device changes IP

---

route

```
route print              # show full routing table
route print -4           # IPv4 only
route print -6           # IPv6 only

# Add a route (non-persistent, lost after reboot)
route add 10.0.0.0 mask 255.0.0.0 192.168.1.254

# Add a persistent route (survives reboot)
route -p add 10.0.0.0 mask 255.0.0.0 192.168.1.254

# Add with metric
route add 10.0.0.0 mask 255.0.0.0 192.168.1.254 metric 10

# Delete a route
route delete 10.0.0.0 mask 255.0.0.0

# Change an existing route
route change 10.0.0.0 mask 255.0.0.0 192.168.1.253

# Add route for a specific interface
route add 10.0.0.0 mask 255.0.0.0 192.168.1.254 IF 4   # IF = interface index from route print
```

Reading `route print` output:

The output has multiple sections:

1. **Interface List:** Maps interface index numbers to adapter names/MACs
2. **IPv4 Route Table → Active Routes:** The actual routing table
3. **Persistent Routes:** Routes flagged with `-p`

```
Active Routes:
Network Destination  Netmask          Gateway       Interface    Metric
          0.0.0.0    0.0.0.0      192.168.1.1    192.168.1.50       25
        127.0.0.0    255.0.0.0         On-link       127.0.0.1      306
      192.168.1.0    255.255.255.0     On-link    192.168.1.50       281
    192.168.1.50    255.255.255.255    On-link    192.168.1.50       281
   192.168.1.255    255.255.255.255    On-link    192.168.1.50       281
```

- `On-link` = directly connected, no next-hop router needed
- `0.0.0.0/0.0.0.0` = default route
- Metric is the route cost — lower wins when multiple routes match the same destination
- A `/32` (`255.255.255.255`) host route for your own IP is normal
- A `/32` for the broadcast address is normal

Troubleshooting with route: If traffic goes to the wrong interface, check `route print` and look for unexpected specific routes that override the default. VPN clients often inject host routes.

PART 4: LINUX NETWORKING — COMPLETE COMMAND REFERENCE

ip Command (Modern — Replaces ifconfig/route/arp)

The `ip` command from the `iproute2` package is the authoritative modern Linux networking tool. It's object-oriented: `ip [object] [command] [options]`.

Objects: `addr`, `link`, `route`, `neigh`, `rule`, `tunnel`, `monitor`, etc.

ip addr — IP Address Management

```
ip addr show                    # show all interfaces and their addresses
ip addr show dev eth0           # show only eth0
ip addr show up                 # show only interfaces that are up
ip -4 addr show                 # IPv4 only
ip -6 addr show                 # IPv6 only
ip -brief addr show             # condensed one-line-per-interface output
```

Reading `ip addr show` output:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether aa:bb:cc:dd:ee:ff brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.50/24 brd 192.168.1.255 scope global dynamic eth0
       valid_lft 86350sec preferred_lft 86350sec
    inet6 fe80::1234:5678:abcd:ef01/64 scope link
       valid_lft forever preferred_lft forever
```

- `<BROADCAST,MULTICAST,UP,LOWER_UP>`: Interface flags. `UP` = interface enabled. `LOWER_UP` = physical link is active (cable connected). `NO-CARRIER` means no link.
- `mtu 1500`: Maximum Transmission Unit in bytes
- `qdisc fq_codel`: Queue discipline (how outgoing packets are scheduled)
- `state UP`: Interface state
- `inet 192.168.1.50/24`: IPv4 address with CIDR prefix
- `scope global`: Routable globally (`link` = link-local only, `host` = loopback only)
- `dynamic`: Address was assigned by DHCP
- `valid_lft` / `preferred_lft`: DHCP lease time remaining

```
# Add an IP address
ip addr add 192.168.1.50/24 dev eth0

# Add with broadcast explicitly specified
ip addr add 192.168.1.50/24 broadcast 192.168.1.255 dev eth0

# Add multiple IPs to same interface (IP aliasing)
ip addr add 10.0.0.1/24 dev eth0
ip addr add 10.0.0.2/24 dev eth0   # both exist simultaneously

# Delete an IP address
ip addr del 192.168.1.50/24 dev eth0

# Flush all addresses from an interface
ip addr flush dev eth0
```

> ***Note:*** *Changes made with* `ip addr` *are not persistent across reboots. For persistence, configure via NetworkManager,* `/etc/network/interfaces`*, or* `systemd-networkd`*.*

---

ip link — Interface / Link Management

```
ip link show                       # all interfaces
ip link show dev eth0              # specific interface
ip -brief link show                # condensed output
```

Reading `ip link show`**:**

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP qlen 1000
    link/ether aa:bb:cc:dd:ee:ff brd ff:ff:ff:ff:ff:ff
```

```
# Bring interface up/down
ip link set eth0 up
ip link set eth0 down

# Change MTU
ip link set eth0 mtu 9000    # jumbo frames (if switch supports it)

# Change MAC address (must be down first)
ip link set eth0 down
ip link set eth0 address 02:11:22:33:44:55
ip link set eth0 up

# Enable/disable promiscuous mode (receive all frames, not just your MAC)
ip link set eth0 promisc on
ip link set eth0 promisc off

# Set TX queue length (default 1000 for Ethernet)
ip link set eth0 txqueuelen 10000

# Set interface alias (human-readable description)
ip link set eth0 alias "WAN Interface"

# Create a VLAN interface (VLAN 100 on eth0)
ip link add link eth0 name eth0.100 type vlan id 100
ip link set eth0.100 up

# Create a bridge
ip link add name br0 type bridge
ip link set eth0 master br0

# Create a dummy interface (useful for testing)
ip link add dummy0 type dummy
```

---

ip route — Routing Table Management

```
ip route show                    # show main routing table
ip route show table all          # show all routing tables
ip -4 route show                 # IPv4 only
ip -6 route show                 # IPv6 only
```

Reading `ip route show` output:

```
default via 192.168.1.1 dev eth0 proto dhcp metric 100
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.50 metric 100
```

- `default via 192.168.1.1`: Default gateway
- `proto dhcp`: This route was installed by the DHCP client
- `proto kernel`: This route was installed by the kernel (connected network)
- `scope link`: Directly reachable without a gateway
- `src 192.168.1.50`: Preferred source address when sending from this interface
- `metric 100`: Route metric/cost

```
# Add default route (gateway)
ip route add default via 192.168.1.1

# Add default route via specific interface
ip route add default via 192.168.1.1 dev eth0

# Add specific network route
ip route add 10.0.0.0/8 via 192.168.1.254

# Add route with metric
ip route add 10.0.0.0/8 via 192.168.1.254 metric 50

# Delete a route
ip route del default via 192.168.1.1
ip route del 10.0.0.0/8

# Change an existing route
ip route change 10.0.0.0/8 via 192.168.1.253

# Replace a route (add if not exists, change if exists)
ip route replace 10.0.0.0/8 via 192.168.1.254

# Show which route will be used for a destination
ip route get 8.8.8.8
# Output: 8.8.8.8 via 192.168.1.1 dev eth0 src 192.168.1.50
```

---

ip neigh — ARP/Neighbor Table

```
ip neigh show                    # show ARP table (IPv4) and neighbor table (IPv6)
ip neigh show dev eth0           # for specific interface
ip -4 neigh show                 # IPv4 ARP only
ip -6 neigh show                 # IPv6 neighbors only
```

Reading output:

```
192.168.1.1 dev eth0 lladdr aa:bb:cc:dd:ee:ff REACHABLE
192.168.1.100 dev eth0 lladdr 11:22:33:44:55:66 STALE
192.168.1.200 dev eth0  FAILED
```

States:

- `REACHABLE`: Entry is fresh and verified reachable
- `STALE`: Entry exists but hasn't been verified recently; will be verified on next use
- `DELAY`: Awaiting confirmation (triggered by using a STALE entry)
- `PROBE`: Sending ARP request to verify
- `FAILED`: ARP requests went unanswered; device probably offline
- `PERMANENT`: Statically configured

```
# Add a static ARP entry
ip neigh add 192.168.1.100 lladdr aa:bb:cc:dd:ee:ff dev eth0

# Change an existing entry
ip neigh change 192.168.1.100 lladdr 11:22:33:44:55:66 dev eth0

# Delete an entry
ip neigh del 192.168.1.100 dev eth0

# Flush all dynamic entries for an interface
ip neigh flush dev eth0

# Flush all stale/failed entries
ip neigh flush nud stale
ip neigh flush nud failed
```

---

iw Command — Wireless Interface Tool

`iw` is the modern nl80211-based wireless configuration tool. Requires the interface to support `cfg80211`.

```
# Show all wireless interfaces
iw dev

# Show capabilities of all wireless PHYs
iw list          # very verbose; shows supported bands, modes, rates, features

# Show specific interface info
iw dev wlan0 info
```

Reading `iw dev wlan0 info` output:

```
Interface wlan0
    ifindex 3
    wdev 0x1
    addr aa:bb:cc:dd:ee:ff
    ssid MyNetwork
    type managed
    wiphy 0
    channel 6 (2437 MHz), width: 20 MHz, center1: 2437 MHz
    txpower 20.00 dBm
```

Scanning for Networks

```
# Scan (must be run as root)
sudo iw dev wlan0 scan

# Condensed scan output
sudo iw dev wlan0 scan | grep -E "SSID:|signal:|freq:|BSS"

# Scan on a specific frequency
sudo iw dev wlan0 scan freq 2412
```

Key scan output fields:

- `BSS aa:bb:cc:dd:ee:ff` — BSSID of the AP
- `freq: 5180` — frequency in MHz (5180 = channel 36 on 5 GHz)
- `signal: -65.00 dBm` — signal strength in dBm (closer to 0 = stronger; -65 is decent; below -80 is poor)
- `SSID: MyNetwork` — network name
- `capability: ESS` — Extended Service Set (infrastructure mode)
- `RSN:` section — security info (WPA2/WPA3, cipher suites)

Setting Interface Mode

```
# Set to managed mode (normal client mode)
sudo ip link set wlan0 down
sudo iw dev wlan0 set type managed
sudo ip link set wlan0 up

# Set to monitor mode (promiscuous — receives all 802.11 frames)
sudo ip link set wlan0 down
sudo iw dev wlan0 set type monitor
sudo ip link set wlan0 up

# Create a new monitor mode interface without bringing existing one down
sudo iw dev wlan0 interface add mon0 type monitor
```

Monitor mode is used for:

- Capturing Wi-Fi traffic with Wireshark/tcpdump
- Passive network discovery
- Frame-level debugging
- Not all drivers support monitor mode (use `iw list` and look for "supported interface modes")

Setting Channel

```
# Set channel (interface must be down or in monitor mode)
sudo iw dev wlan0 set channel 6
sudo iw dev wlan0 set channel 36 HT40+    # 40 MHz, extension channel above
sudo iw dev wlan0 set channel 36 HT40-    # 40 MHz, extension channel below
sudo iw dev wlan0 set channel 36 80MHz    # 80 MHz width (if supported)
```

Transmit Power

```
# Show current txpower
iw dev wlan0 info | grep txpower

# Set txpower (value in mBm = millidecibel-milliwatts, 100 * dBm)
sudo iw dev wlan0 set txpower fixed 1500    # 15 dBm
sudo iw dev wlan0 set txpower limit 2000    # max 20 dBm
sudo iw dev wlan0 set txpower auto          # let driver decide

# Set on phy level
sudo iw phy phy0 set txpower fixed 1500
```

Regulatory Domain

```
# Show current regulatory domain
iw reg get

# Set regulatory domain (affects allowed channels and power limits)
sudo iw reg set US     # United States
sudo iw reg set DE     # Germany
sudo iw reg set BO     # Bolivia (notorious for allowing high power and extra channels)
```

Why regulatory domain matters: Different countries allow different maximum transmit power and different channels. Reg domain `BO` (Bolivia) is commonly used to unlock DFS channels and higher power in pentesting contexts. Changing reg domain without proper authorization may violate law.

---

iwconfig (Legacy — Still Encountered)

`iwconfig` is the legacy wireless tool from `wireless-tools`. Still present on most systems and commonly seen in tutorials.

```
iwconfig              # show wireless info for all interfaces
iwconfig wlan0        # specific interface

# Set SSID
iwconfig wlan0 essid "MyNetwork"

# Set channel (AP mode)
iwconfig wlan0 channel 6

# Set mode
iwconfig wlan0 mode managed   # client
iwconfig wlan0 mode monitor   # monitor

# Set transmit power (in dBm or mW)
iwconfig wlan0 txpower 20     # 20 dBm
iwconfig wlan0 txpower 100mW  # 100 milliwatts

# Set bit rate
iwconfig wlan0 rate 54M
iwconfig wlan0 rate auto
```

Reading `iwconfig wlan0` output:

```
wlan0  IEEE 802.11  ESSID:"MyNetwork"
       Mode:Managed  Frequency:5.18 GHz  Access Point: AA:BB:CC:DD:EE:FF
       Bit Rate=300 Mb/s   Tx-Power=20 dBm
       Retry short limit:7   RTS thr:off   Fragment thr:off
       Power Management:on
       Link Quality=65/70  Signal level=-45 dBm  Noise level=-95 dBm
       Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
       Tx excessive retries:0  Invalid misc:0  Missed beacon:0
```

- `Link Quality=65/70`: 65/70 = 93% — qualitative score
- `Signal level=-45 dBm`: RSSI (Received Signal Strength Indicator). Excellent = above -50, Good = -50 to -60, Fair = -60 to -70, Poor = below -70
- `Noise level=-95 dBm`: Background noise floor
- SNR = Signal - Noise = -45 - (-95) = 50 dB → Excellent
- `Tx excessive retries`: Count of frames that needed many retransmissions — high numbers indicate poor link quality
- `Missed beacon`: Count of missed AP beacons — high = AP is far/unreachable

`iw` vs `iwconfig`: Use `iw` for new work. `iwconfig` is deprecated but still works. On some older distributions or embedded devices (OpenWRT), `iwconfig` may be the only tool available.

---

nmcli — NetworkManager CLI

`nmcli` controls NetworkManager, which is the connection manager used by most modern desktop Linux distros (Ubuntu, Fedora, etc.). It manages "connections" (saved network profiles) and "devices" (physical interfaces).

Device Status

```
nmcli device status             # all devices: type, state, connection
nmcli device show               # detailed info for all devices
nmcli device show wlan0        # detailed info for wlan0
nmcli device wifi list          # scan for and list Wi-Fi networks
nmcli device wifi list ifname wlan0    # specify interface
```

Reading `nmcli device wifi list`:

```
IN-USE  BSSID              SSID          MODE   CHAN  RATE        SIGNAL  BARS  SECURITY
*       AA:BB:CC:DD:EE:FF  MyNetwork     Infra  6     130 Mbit/s  85      ▂▄▆█  WPA2
        11:22:33:44:55:66  Neighbor      Infra  11    54 Mbit/s   40      ▂▄__  WPA1
```

- `IN-USE` (`*`): Currently connected
- `SIGNAL 85`: 0–100 scale, not dBm (NetworkManager converts)
- `RATE`: Maximum rate supported by the AP

Connect to Wi-Fi

```
# Connect to a network (creates a new connection profile)
nmcli device wifi connect "MyNetwork" password "mypassword"

# Connect on specific interface
nmcli device wifi connect "MyNetwork" password "mypassword" ifname wlan0

# Connect to hidden network
nmcli device wifi connect "HiddenNet" password "pass" hidden yes

# Connect to a specific BSSID
nmcli device wifi connect "MyNetwork" bssid AA:BB:CC:DD:EE:FF password "pass"
```

Connection Management

```
nmcli connection show                    # list all saved connections
nmcli connection show --active           # currently active ones only
nmcli connection show "MyNetwork"        # details for a specific connection

# Bring a connection up/down
nmcli connection up "MyNetwork"
nmcli connection down "MyNetwork"

# Delete a connection profile
nmcli connection delete "OldNetwork"
```

Configure Static IP

```
# Modify an existing connection to static IP
nmcli connection modify "MyNetwork" \
    ipv4.method manual \
    ipv4.addresses "192.168.1.50/24" \
    ipv4.gateway "192.168.1.1" \
    ipv4.dns "8.8.8.8,1.1.1.1"

# Apply changes
nmcli connection up "MyNetwork"
```

Configure Multiple DNS Servers, Search Domains

```
nmcli connection modify "MyNetwork" ipv4.dns "8.8.8.8 1.1.1.1"
nmcli connection modify "MyNetwork" ipv4.dns-search "corp.example.com example.com"
```

Set Back to DHCP

```
nmcli connection modify "MyNetwork" ipv4.method auto
nmcli connection modify "MyNetwork" ipv4.addresses ""
nmcli connection modify "MyNetwork" ipv4.gateway ""
nmcli connection modify "MyNetwork" ipv4.dns ""
nmcli connection up "MyNetwork"
```

MAC Randomization

```
nmcli connection modify "MyNetwork" wifi.cloned-mac-address random
nmcli connection modify "MyNetwork" wifi.cloned-mac-address stable    # consistent per SSID
nmcli connection modify "MyNetwork" wifi.cloned-mac-address permanent # use real HW MAC
```

Radio Control

```
nmcli radio wifi on
nmcli radio wifi off
nmcli radio all off     # airplane mode equivalent
nmcli radio all on
```

---

ifconfig (Legacy)

Still present on most systems via `net-tools`. Commonly seen in tutorials and on older/embedded systems.

```
ifconfig                  # show all active interfaces
ifconfig -a               # show all interfaces including down ones
ifconfig eth0             # show specific interface

# Set IP and netmask
ifconfig eth0 192.168.1.50 netmask 255.255.255.0

# Bring up/down
ifconfig eth0 up
ifconfig eth0 down

# Set MTU
ifconfig eth0 mtu 1400

# Set MAC (must be down first)
ifconfig eth0 hw ether 02:11:22:33:44:55

# Add secondary IP (alias)
ifconfig eth0:1 10.0.0.1 netmask 255.255.255.0
```

Reading `ifconfig eth0` output:

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.50  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::aabb:ccff:fedd:eeff  prefixlen 64  scopeid 0x20<link>
        ether aa:bb:cc:dd:ee:ff  txqueuelen 1000  (Ethernet)
        RX packets 12345  bytes 9876543 (9.4 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11111  bytes 8765432 (8.4 MiB)
        TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
```

- `RX errors`: Received frames with errors (CRC, framing, etc.) — indicates physical link issues
- `dropped`: Frames dropped due to no buffer space — NIC ring buffer overrun; check for CPU bottleneck
- `overruns`: Ring buffer overflows — system couldn't process packets fast enough
- `carrier`: Carrier signal loss count — physical layer issues
- `collisions`: Should be 0 on full-duplex links. Non-zero means half-duplex or duplex mismatch.

---

netstat / ss

`ss` (socket statistics) is the modern replacement for `netstat`. Both show open ports and connections, but `ss` is faster (reads directly from kernel) and shows more detail.

Common Options (Both Commands)

Flag	Meaning
`-t`	TCP sockets
`-u`	UDP sockets
`-l`	Listening sockets only
`-p`	Show process name/PID (requires root for other users' processes)
`-n`	Numeric (don't resolve hostnames/service names)
`-a`	All sockets (listening + established)
`-4`	IPv4 only (ss)
`-6`	IPv6 only (ss)
`-e`	Extended info (ss)
`-s`	Summary statistics

```
# Most useful: all listening TCP/UDP ports with process info
ss -tulpn
netstat -tulpn    # equivalent but slower

# All connections (active + listening)
ss -tan
netstat -tan

# Show specific port
ss -tlnp sport = :80    # services listening on port 80
ss -tnp dst :443        # connections to port 443

# Show connections from a specific address
ss -tn src 192.168.1.50

# Extended info with timers
ss -to

# Socket memory usage
ss -m

# Established connections only
ss -tn state established
```

Reading `ss -tulpn` output:

```
Netid  State   Recv-Q  Send-Q  Local Address:Port  Peer Address:Port  Process
tcp    LISTEN  0       128     0.0.0.0:22          0.0.0.0:*          users:(("sshd",pid=1234,fd=3))
tcp    LISTEN  0       511     127.0.0.1:5432      0.0.0.0:*          users:(("postgres",pid=5678,fd=5
udp    UNCONN  0       0       0.0.0.0:68          0.0.0.0:*          users:(("dhclient",pid=910,fd=7)
```

- `0.0.0.0:22`: Listening on all interfaces, port 22 (SSH) — reachable from anywhere
- `127.0.0.1:5432`: PostgreSQL listening only on loopback — local connections only, not network-accessible
- `Recv-Q`: Bytes in receive queue. Non-zero on LISTEN socket = backlog accumulating (server slow to accept)
- `Send-Q`: Bytes waiting to be sent. Non-zero = congestion or slow receiver
- `UNCONN`: UDP has no connection state; this is normal for UDP servers

---

ping (Linux)

```
ping google.com                   # continuous until Ctrl+C (unlike Windows)
ping -c 4 google.com              # send exactly 4 packets
ping -i 0.2 google.com            # interval 0.2s between packets
ping -s 1400 google.com           # packet size 1400 bytes
ping -M do google.com             # Don't Fragment bit (equivalent of Windows -f)
ping -t 5 google.com              # TTL = 5
ping -W 1 google.com              # 1 second timeout per reply
ping -q google.com                # quiet, only show summary
ping -v google.com                # verbose
ping -6 google.com                # force IPv6
ping6 google.com                  # IPv6 ping on older systems
ping -b 192.168.1.255             # broadcast ping (find live hosts)
ping -I eth0 google.com           # use specific interface
ping -I 192.168.1.50 google.com   # use specific source IP
```

MTU discovery with ping:

```
# Find MTU: send unfragmentable packets, decrease size until it works
ping -M do -s 1472 google.com   # 1472 + 28 header = 1500 bytes
ping -M do -s 1400 google.com   # if 1472 fails, try smaller
```

---

traceroute

```
traceroute google.com
traceroute -n 8.8.8.8            # no hostname resolution
traceroute -m 20 google.com      # max 20 hops
traceroute -w 2 google.com       # 2 second wait per probe
traceroute -q 1 google.com       # send only 1 probe per hop (faster, less info)
traceroute -I google.com         # use ICMP Echo (like Windows tracert)
traceroute -T google.com         # use TCP SYN (better firewall penetration)
traceroute -T -p 443 google.com  # TCP traceroute to port 443
traceroute -U -p 53 8.8.8.8     # UDP to port 53
```

Default behavior: Linux `traceroute` uses UDP (unlike Windows tracert which uses ICMP). UDP probes are sent to high-number ports (33434+). ICMP mode (`-I`) is needed for most firewall comparisons with Windows.

TCP traceroute (`-T`) is useful in enterprise environments where ICMP/UDP is blocked by firewalls but TCP/443 is open. You can trace the path that actual HTTPS traffic would take.

---

mtr — My Traceroute (Better Traceroute)

`mtr` combines traceroute and ping. It continuously probes each hop and shows packet loss and jitter in real time.

```
mtr google.com                    # interactive live display
mtr --report google.com          # run 10 cycles, print report, exit
mtr --report-cycles 50 google.com  # 50 cycles
mtr -n google.com                 # no hostname resolution
mtr --tcp -P 443 google.com      # TCP mode, port 443
mtr --udp google.com              # UDP mode
mtr --interval 0.5 google.com    # 500ms between probes
```

Reading mtr output:

```
HOST: mybox                       Loss%   Snt   Last   Avg  Best  Wrst StDev
  1.|-- 192.168.1.1               0.0%    10    1.2    1.1  0.9   1.5   0.2
  2.|-- 10.0.0.1                  0.0%    10   11.3   11.0  10.5  12.1  0.5
  3.|-- 72.14.215.165             0.0%    10   12.5   12.8  11.9  14.1  0.7
  4.|-- ???                       100.0%   10   0.0    0.0   0.0   0.0   0.0
  5.|-- 142.250.80.46              0.0%    10   13.1   13.0  12.5  13.8  0.4
```

- `Loss%`: Packet loss to that hop across all probes so far
- `Snt`: Probes sent
- `Last/Avg/Best/Wrst`: RTT statistics in ms
- `StDev`: Jitter (standard deviation of RTT)
- `???` at hop 4: That router doesn't respond to ICMP but traffic passes through (hop 5 is still reachable)
- 100% loss at hop N but 0% at hop N+1: That specific router drops ICMP TTL-exceeded messages. Not a real problem.
- 100% loss from hop N through the end: Real loss.

---

dig — DNS Query Tool

```
dig google.com                      # A record lookup
dig google.com A                    # explicit type
dig google.com AAAA                 # IPv6
dig google.com MX                   # mail servers
dig google.com NS                   # nameservers
dig google.com TXT                  # text records
dig google.com ANY                  # all records (often filtered)
dig -x 142.250.80.46                # reverse lookup (PTR)
dig @8.8.8.8 google.com             # query specific server
dig @8.8.8.8 google.com +short      # short answer only (just IP)
dig google.com +trace               # trace full resolution chain from root
dig google.com +dnssec              # show DNSSEC records
dig google.com +noall +answer       # show only the answer section
dig google.com +stats               # show query statistics (timing)
dig +time=2 +tries=1 google.com     # 2s timeout, 1 retry
dig -p 5353 @224.0.0.251 google.local # mDNS lookup
```

Reading `dig` output:

```
; <<>> DiG 9.16 <<>> google.com
;; QUESTION SECTION:
;google.com.           IN  A

;; ANSWER SECTION:
google.com.     300    IN  A  142.250.80.46

;; QUERY TIME: 15 msec
;; SERVER: 192.168.1.1#53
;; WHEN: ...
;; MSG SIZE rcvd: 55
```

- `300`: TTL remaining in seconds
- `SERVER: 192.168.1.1#53`: Which DNS server answered
- `QUERY TIME: 15 msec`: Total round-trip time

`dig +trace` **output:** Shows the entire recursive resolution chain from the root servers down. Invaluable for debugging DNS delegation issues.

---

iptables / nftables Basics

How Packet Filtering Works

Packets entering, leaving, or transiting a Linux machine pass through **chains**. Each chain contains ordered **rules**. The first matching rule wins; if nothing matches, the **default policy** applies (ACCEPT or DROP).

Tables:

- `filter`: The default table. Contains INPUT, OUTPUT, FORWARD chains.
- `nat`: For NAT operations. Contains PREROUTING, POSTROUTING, OUTPUT.
- `mangle`: Modifies packet fields (TTL, QoS bits, etc.)
- `raw`: Pre-connection-tracking processing

Chains in filter table:

- `INPUT`: Packets destined for this machine
- `OUTPUT`: Packets originating from this machine
- `FORWARD`: Packets passing through this machine (routing between interfaces)

iptables

```
# Show current rules
iptables -L                          # list filter table rules
iptables -L -n -v                    # verbose with packet counts, numeric
iptables -L -n -v --line-numbers     # show rule numbers (needed for deletion)
iptables -t nat -L -n -v             # list NAT table

# View a specific chain
iptables -L INPUT -n -v

# Allow established/related connections (essential — don't lock yourself out)
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

# Allow loopback
iptables -A INPUT -i lo -j ACCEPT

# Allow SSH (before setting default DROP)
iptables -A INPUT -p tcp --dport 22 -j ACCEPT

# Set default policy to DROP (after adding allow rules)
iptables -P INPUT DROP
iptables -P FORWARD DROP

# Allow pings
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT

# Allow specific source IP
iptables -A INPUT -s 192.168.1.100 -j ACCEPT

# Allow port range
iptables -A INPUT -p tcp --dport 8000:8080 -j ACCEPT

# Block an IP
iptables -A INPUT -s 10.0.0.5 -j DROP

# NAT / IP masquerading (for internet sharing/routing)
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
echo 1 > /proc/sys/net/ipv4/ip_forward

# Port forwarding
iptables -t nat -A PREROUTING -p tcp --dport 8080 -j REDIRECT --to-port 80

# Delete a rule by line number
iptables -L INPUT --line-numbers
iptables -D INPUT 3    # delete rule 3

# Flush all rules (careful — will drop all filtering)
iptables -F
iptables -t nat -F

# Save/restore rules
iptables-save > /etc/iptables/rules.v4
iptables-restore < /etc/iptables/rules.v4
```

nftables (Replaces iptables on modern systems)

```
# Show all rules
nft list ruleset

# Basic rule structure
nft add table inet filter
nft add chain inet filter input { type filter hook input priority 0 \; policy drop \; }
nft add rule inet filter input ct state established,related accept
nft add rule inet filter input iif lo accept
nft add rule inet filter input tcp dport 22 accept
nft add rule inet filter input ip protocol icmp accept

# List with rule handles (for deletion)
nft list ruleset -a

# Delete a rule by handle
nft delete rule inet filter input handle 5

# Save
nft list ruleset > /etc/nftables.conf
```

Network Configuration Files

`/etc/network/interfaces` (Debian/Ubuntu traditional)

```
# /etc/network/interfaces

# Loopback
auto lo
iface lo inet loopback

# DHCP
auto eth0
iface eth0 inet dhcp

# Static IP
auto eth0
iface eth0 inet static
    address 192.168.1.50
    netmask 255.255.255.0
    gateway 192.168.1.1
    dns-nameservers 8.8.8.8 8.8.4.4
    dns-search corp.example.com

# Apply changes
sudo ifdown eth0 && sudo ifup eth0
```

NetworkManager (systemd-based distros)

NetworkManager stores connections as files in `/etc/NetworkManager/system-connections/`. They're INI-format `.nmconnection` files.

```
[connection]
id=MyEthernet
type=ethernet
interface-name=eth0

[ipv4]
method=manual
addresses=192.168.1.50/24
gateway=192.168.1.1
dns=8.8.8.8;1.1.1.1;

[ipv6]
method=auto
```

After editing: `nmcli connection reload`

systemd-networkd

Used on headless servers and some distros (Arch, recent Ubuntu Server). Configuration in `/etc/systemd/network/`.

```
# /etc/systemd/network/20-eth0.network

[Match]
Name=eth0

[Network]
Address=192.168.1.50/24
Gateway=192.168.1.1
DNS=8.8.8.8
DNS=1.1.1.1
Domains=corp.example.com
```

```
# Apply
systemctl restart systemd-networkd
systemctl enable systemd-networkd

# Check status
networkctl
networkctl status eth0
```

/etc/resolv.conf

Controls DNS resolution. Read by `glibc` resolver and thus by most applications.

```
# /etc/resolv.conf

nameserver 8.8.8.8         # Primary DNS server (queried first)
nameserver 1.1.1.1         # Secondary DNS server (fallback)
search corp.example.com    # Append this to unqualified hostnames
domain corp.example.com    # Single-domain equivalent of search (deprecated)
options ndots:5            # Names with fewer than 5 dots tried with search domains first
options timeout:2          # 2-second query timeout
options attempts:3         # Retry 3 times before failing
```

Why it gets overwritten: On systems using `systemd-resolved`, NetworkManager, or dhclient, these tools overwrite `/etc/resolv.conf` on network events. The file is often a symlink to a dynamically generated version.

```
ls -la /etc/resolv.conf    # check if it's a symlink
# Often: /etc/resolv.conf -> /run/systemd/resolve/stub-resolv.conf
```

Making DNS persistent:

Option 1 — Via NetworkManager:

```
nmcli connection modify "MyNetwork" ipv4.dns "8.8.8.8 1.1.1.1"
nmcli connection modify "MyNetwork" ipv4.ignore-auto-dns yes    # ignore DHCP-provided DNS
nmcli connection up "MyNetwork"
```

Option 2 — Pin /etc/resolv.conf (brute force, not recommended):

```
chattr +i /etc/resolv.conf    # make immutable (even root can't overwrite)
chattr -i /etc/resolv.conf    # remove immutability
```

Option 3 — Configure systemd-resolved:

```
# /etc/systemd/resolved.conf
[Resolve]
DNS=8.8.8.8 1.1.1.1
FallbackDNS=9.9.9.9
Domains=~.
```

```
systemctl restart systemd-resolved
```

PART 5: MICROCONTROLLER / EMBEDDED NETWORKING

How ESP32 / ESP8266 Handle Wi-Fi (Arduino Framework)

The ESP32 (Espressif) is the dominant embedded Wi-Fi microcontroller for makers and low-volume IoT. The ESP8266 preceded it with a simpler feature set. Both run an 802.11 station (client) and/or access point on a single chip.

Fundamental architecture:

- The Wi-Fi radio runs on a separate co-processor with its own RTOS
- Your Arduino `loop()` runs on the application core
- Wi-Fi events fire callbacks in a separate FreeRTOS task
- You interact with the Wi-Fi stack via the `WiFi` library (ESP8266WiFi.h / WiFi.h on ESP32)

Basic Connection Pattern

```
#include <WiFi.h>    // ESP32
// #include <ESP8266WiFi.h>  // ESP8266

const char* ssid = "MyNetwork";
const char* password = "mypassword";

void setup() {
    Serial.begin(115200);
    WiFi.mode(WIFI_STA);           // station mode (client)
    WiFi.begin(ssid, password);

    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println();
    Serial.print("Connected. IP: ");
    Serial.println(WiFi.localIP());
}
```

WiFi status codes:

- `WL_IDLE_STATUS (0)`: WiFi is initializing
- `WL_NO_SSID_AVAIL (1)`: SSID not found in scan
- `WL_SCAN_COMPLETED (2)`: Scan finished
- `WL_CONNECTED (3)`: Associated and IP obtained
- `WL_CONNECT_FAILED (4)`: Association failed (wrong password or rejection)
- `WL_CONNECTION_LOST (5)`: Connection dropped
- `WL_DISCONNECTED (6)`: Not connected

Setting a Static IP on ESP32

This must be called **before** `WiFi.begin()`. The IP configuration goes into the Wi-Fi driver before the DHCP client attempts to run.

```
#include <WiFi.h>

IPAddress local_IP(192, 168, 1, 100);    // desired static IP
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 255, 0);
IPAddress primaryDNS(8, 8, 8, 8);        // optional
IPAddress secondaryDNS(8, 8, 4, 4);      // optional

void setup() {
    WiFi.mode(WIFI_STA);

    // Configure static IP BEFORE WiFi.begin()
    if (!WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS)) {
        Serial.println("STA Failed to configure");
    }

    WiFi.begin("MyNetwork", "mypassword");

    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
    }
    Serial.println(WiFi.localIP());    // should print 192.168.1.100
}
```

Why this works: When `WiFi.config()` is called, it tells the underlying lwIP stack to skip DHCP and use the static parameters. If `WiFi.config()` is not called, DHCP is enabled by default.

Scanning for Networks

```
void scanNetworks() {
    Serial.println("Scanning...");
    int n = WiFi.scanNetworks();    // blocking scan; returns number found

    if (n == 0) {
        Serial.println("No networks found");
    } else {
        for (int i = 0; i < n; i++) {
            Serial.print(i + 1);
            Serial.print(": ");
            Serial.print(WiFi.SSID(i));           // SSID
            Serial.print(" (");
            Serial.print(WiFi.RSSI(i));           // Signal strength in dBm
            Serial.print(" dBm) ");
            Serial.print(WiFi.channel(i));        // Channel
            Serial.print(" Ch ");
            Serial.print(WiFi.encryptionType(i) == WIFI_AUTH_OPEN ? "Open" : "Secured");
            Serial.println();
        }
    }
    WiFi.scanDelete();    // free scan memory
}
```

Async scan (non-blocking):

```
WiFi.scanNetworks(true);    // async = true

// Later, check:
int n = WiFi.scanComplete();
if (n >= 0) {
    // process results
} else if (n == WIFI_SCAN_RUNNING) {
    // still scanning
}
```

802.11 Association Process at the Packet Level

Understanding this is essential for interpreting monitor-mode captures:

Phase 1: Discovery

1. **Passive scanning:** Client listens on each channel for beacon frames (AP broadcasts every ~100ms)
2. **Active scanning:** Client sends **Probe Request** frames (optionally with desired SSID or broadcast SSID=wildcard). APs respond with **Probe Response** containing full capabilities.

Phase 2: Authentication (802.11 Open or SAE)

- **Open System Authentication:** Client sends Auth Request → AP sends Auth Response (always success). This is just a formality in modern WPA2/WPA3; actual credential checking happens in the next phase.
- **WPA3 SAE:** Replaces this with Commit + Confirm messages for the Dragonfly handshake.

Phase 3: Association

- Client sends **Association Request** to chosen AP (BSSID): "I want to join, I support these rates and capabilities"
- AP sends **Association Response**: Assigns Association ID (AID), confirms capabilities

Phase 4: EAP / 4-Way Handshake (WPA2-Personal)

1. AP sends ANonce (random number)
2. Client generates PTK (Pairwise Transient Key) from ANonce + SNonce + PMK (PMK derived from passphrase via PBKDF2-SHA1 with 4096 iterations)
3. Client sends SNonce + MIC (integrity check)
4. AP verifies, installs PTK, sends GTK (Group Temporal Key, for broadcast/multicast)
5. Client sends ACK

The handshake is what you capture for offline password cracking. The MIC in message 2 or 3 is computed using the PTK which is derived from the passphrase. You capture messages 1+2 or 2+3, then brute-force the passphrase by recomputing the PTK for each candidate and checking if the MIC matches.

Relevant fields in captures:

- Frame type 0 = Management frames (beacons, probes, auth, assoc)
- Frame type 1 = Control frames (RTS, CTS, ACK)
- Frame type 2 = Data frames (actual data)
- `wlan.fc.type_subtype == 0x08` = Beacon
- `wlan.fc.type_subtype == 0x04` = Probe Request
- `eapol` filter in Wireshark = show 4-way handshake frames

TCP vs UDP — When to Use Which, Socket Basics

TCP:

- Connection-oriented (three-way handshake: SYN, SYN-ACK, ACK)
- Guaranteed delivery (automatic retransmission of lost packets)
- Ordered (receiver reconstructs original order)
- Flow control (sender doesn't overwhelm receiver)
- Congestion control (backs off when network is congested)
- Full-duplex
- **Overhead:** ~20-byte header + handshake time + ACK round trips

UDP:

- Connectionless (no handshake)
- Best-effort delivery (fire and forget)
- No ordering guarantee
- No flow or congestion control
- **Overhead:** 8-byte header only
- Lower latency (no waiting for ACK before next packet)

When to use TCP on embedded:

- Sending data that must arrive (configuration commands, firmware OTA, sensor data that can't be lost)

- HTTP/HTTPS (always TCP)
- MQTT (by default uses TCP — it adds its own QoS layers on top)

When to use UDP on embedded:

- Time-stamped telemetry where a missing packet is better to skip than wait for (sensor readings every 100ms — old data is useless)
- DNS queries (short, single-packet request/response; UDP is faster for this)
- CoAP (Constrained Application Protocol) — designed for IoT, uses UDP
- NTP (time sync — single request/response, timing is critical)
- Multicast discovery

Socket basics on ESP32:

TCP client:

```
#include <WiFiClient.h>

WiFiClient client;
if (client.connect("192.168.1.100", 80)) {    // TCP connect
    client.print("GET / HTTP/1.0\r\nHost: 192.168.1.100\r\n\r\n");
    while (client.available()) {
        String line = client.readStringUntil('\n');
        Serial.println(line);
    }
    client.stop();
}
```

UDP:

```
#include <WiFiUdp.h>

WiFiUdp udp;
udp.begin(4210);    // listen on local port 4210

// Send
udp.beginPacket("192.168.1.100", 4210);
udp.write("hello");
udp.endPacket();

// Receive (poll in loop)
int packetSize = udp.parsePacket();
if (packetSize) {
    char buf[255];
    int len = udp.read(buf, 255);
    buf[len] = 0;
    Serial.println(buf);
}
```

HTTP vs MQTT vs Raw TCP for IoT

HTTP/HTTPS:

- Request-response model. Client initiates every exchange.
- High overhead per message (headers are large, typically 200–500 bytes)
- Uses TCP (usually port 80/443)
- Stateless: each request is independent (unless keep-alive is used)
- Good for: infrequent data sending to a REST API, receiving configuration on demand
- Bad for: high-frequency telemetry (too much overhead), real-time bidirectional communication, battery-critical devices

ESP32 HTTPS example (using HTTPClient):

```
#include <HTTPClient.h>
#include <WiFiClientSecure.h>

WiFiClientSecure client;
client.setInsecure();    // skip TLS cert verification (NOT for production)
// client.setCACert(root_ca);  // proper cert verification

HTTPClient https;
https.begin(client, "https://api.example.com/data");
int httpCode = https.POST("{\"temp\":23.5}");
String response = https.getString();
https.end();
```

MQTT (Message Queue Telemetry Transport):

- Publish/subscribe model. Devices publish to "topics"; subscribers receive matching messages.
- Very low overhead: minimum 2-byte header for PUBLISH
- Persistent TCP connection with keepalive (ping/pong every N seconds)
- Broker (e.g., Mosquitto, AWS IoT, HiveMQ) handles routing
- QoS levels:
- 0: Fire and forget (UDP-like, over TCP)
- 1: At least once (ACK'd, possible duplicates)
- 2: Exactly once (4-packet exchange)
- Retained messages: broker keeps last message on topic; new subscribers receive it immediately
- Will messages: message broker sends on behalf of a client if it disconnects unexpectedly
- Good for: frequent telemetry, command/control, many-to-many communication, IoT fleets

ESP32 MQTT example (using PubSubClient):

```
#include <PubSubClient.h>
#include <WiFiClient.h>

WiFiClient espClient;
PubSubClient mqtt(espClient);

void callback(char* topic, byte* payload, unsigned int length) {
    // Handle received message
    String msg = "";
    for (int i = 0; i < length; i++) msg += (char)payload[i];
    Serial.println("Received: " + msg);
}

void setup() {
    // ... WiFi connect ...
    mqtt.setServer("192.168.1.10", 1883);    // broker IP and port
    mqtt.setCallback(callback);
}

void loop() {
    if (!mqtt.connected()) {
        mqtt.connect("ESP32-device-01");        // client ID
        mqtt.subscribe("devices/esp32/cmd");    // subscribe to commands
    }
    mqtt.loop();    // must be called regularly to handle incoming messages

    // Publish sensor data
    char payload[64];
    snprintf(payload, sizeof(payload), "{\"temp\":%.1f}", readTemperature());
    mqtt.publish("devices/esp32/telemetry", payload);
    delay(10000);
}
```

Raw TCP:

- Custom binary protocol, maximum efficiency
- You define message framing yourself
- Good for: high-performance custom applications, when you control both ends, streaming data
- Bad for: interoperability, protocol versioning, debugging

DNS Resolution on Microcontrollers

The ESP32 uses **lwIP** (lightweight IP stack) which includes a DNS client. It queries the DNS server provided via DHCP (or static configuration).

```
#include <WiFi.h>

// After connecting to WiFi:
IPAddress resolved;
if (WiFi.hostByName("api.example.com", resolved) == 1) {
    Serial.println(resolved.toString());
} else {
    Serial.println("DNS resolution failed");
}
```

DNS caching on ESP32: lwIP has a minimal DNS cache (controlled by `DNS_TABLE_SIZE`, default 4 entries). If you're hitting many different hostnames, responses get evicted quickly. For devices that always talk to the same server, this is not an issue.

mDNS (multicast DNS): For local network discovery without a DNS server. Used by Apple Bonjour, Avahi on Linux. The ESP32 can both advertise and query mDNS.

```
#include <ESPmDNS.h>

// Advertise this device as "mydevice.local"
MDNS.begin("mydevice");

// Optionally advertise a service
MDNS.addService("http", "tcp", 80);

// Query: find any device advertising an HTTP service
int n = MDNS.queryService("http", "tcp");
for (int i = 0; i < n; i++) {
    Serial.println(MDNS.hostname(i));   // hostname
    Serial.println(MDNS.IP(i));         // IP address
    Serial.println(MDNS.port(i));       // port
}
```

DHCP Client Behavior on Embedded Devices

When `WiFi.begin()` is called without `WiFi.config()`, the ESP32's lwIP DHCP client:

1. Sends DHCPDISCOVER via broadcast after association
2. Takes the first DHCPOFFER received
3. Sends DHCPREQUEST
4. Waits for DHCPACK
5. Configures IP, gateway, subnet mask, DNS

The `WiFi.status()` becomes `WL_CONNECTED` only after the IP is obtained, not just after association.

Lease renewal: lwIP handles renewal automatically at 50% of lease time (sends DHCPREQUEST unicast to server). If renewal fails, it attempts broadcast renewal at 87.5%. If the ESP32 is in deep sleep, it won't renew — on wake, the DHCP lease may have expired and it needs a new one. Solutions:

- Use a long DHCP lease time on your router for IoT devices
- Use a DHCP reservation (same IP always assigned to this MAC)

- Configure a static IP in firmware

Reconnection after disconnect:

```
// Recommended pattern for robust Wi-Fi on ESP32
void loop() {
    if (WiFi.status() != WL_CONNECTED) {
        Serial.println("Reconnecting...");
        WiFi.disconnect();
        WiFi.reconnect();
        // or WiFi.begin(ssid, password) if reconnect doesn't work
        int attempts = 0;
        while (WiFi.status() != WL_CONNECTED && attempts < 20) {
            delay(500);
            attempts++;
        }
    }
    // ... rest of loop
}
```

WiFi sleep modes (power vs responsiveness):

```
WiFi.setSleep(true);     // modem sleep — radio sleeps between DTIM intervals
WiFi.setSleep(false);    // no sleep — maximum responsiveness, more power

// Light sleep (available on ESP32 with arduino-esp32 >= 2.x)
esp_wifi_set_ps(WIFI_PS_MIN_MODEM);    // sleep between beacons
esp_wifi_set_ps(WIFI_PS_MAX_MODEM);    // sleep more aggressively
esp_wifi_set_ps(WIFI_PS_NONE);         // no sleep
```

NTP on Embedded Devices

Real-time clocks on ESP32 require NTP since there's no battery-backed RTC.

```
#include <time.h>

void setupTime() {
    configTime(0, 0, "pool.ntp.org", "time.nist.gov");    // UTC offset, DST offset, NTP servers

    // Wait for time to be set
    struct tm timeinfo;
    while (!getLocalTime(&timeinfo)) {
        Serial.println("Waiting for NTP...");
        delay(1000);
    }
    Serial.println(&timeinfo, "Time: %A %B %d %Y %H:%M:%S");
}
```

NTP uses UDP port 123. The ESP32 sends a request to the NTP server, which responds with the current time. The ESP32 doesn't implement full NTP clock discipline (slewing) — it just sets the system clock to the received time.

Practical Network Debugging on Embedded

```
// Print full network info for diagnostics
void printNetworkInfo() {
    Serial.print("SSID: ");         Serial.println(WiFi.SSID());
    Serial.print("BSSID: ");        Serial.println(WiFi.BSSIDstr());
    Serial.print("Channel: ");      Serial.println(WiFi.channel());
    Serial.print("RSSI: ");         Serial.print(WiFi.RSSI()); Serial.println(" dBm");
    Serial.print("IP: ");           Serial.println(WiFi.localIP());
    Serial.print("Subnet: ");       Serial.println(WiFi.subnetMask());
    Serial.print("Gateway: ");      Serial.println(WiFi.gatewayIP());
    Serial.print("DNS: ");          Serial.println(WiFi.dnsIP());
    Serial.print("MAC: ");          Serial.println(WiFi.macAddress());
    Serial.print("Hostname: ");     Serial.println(WiFi.getHostname());
}
```

Common ESP32 Wi-Fi failures and causes:

Symptom	Likely Cause
`WL_CONNECT_FAILED`	Wrong password, AP rejected association
`WL_NO_SSID_AVAIL`	SSID not found — wrong name, out of range, hidden network not configured
Connects then drops immediately	DHCP failure (no server, pool exhausted), or IP conflict
Connects but can't reach internet	Wrong gateway, no DNS, or internet on the router side
Frequent disconnects	Weak signal (check RSSI), RF interference, AP deauths (check AP logs)
Won't connect to 5 GHz	ESP32 original only supports 2.4 GHz; ESP32-S2/S3/C3 vary — check datasheet

PART 6: PRACTICAL SCENARIOS

Scenario 1: Give a Device a Static IP

The goal: a device always has the same IP address, so you can reliably reach it, configure port forwarding, or reference it in other tools.

Method A: Static IP Configured Directly on the Device

Windows (GUI):

```
Control Panel → Network and Sharing Center → Change Adapter Settings
→ Right-click adapter → Properties → Internet Protocol Version 4 (TCP/IPv4)
→ "Use the following IP address"
→ Enter: IP, Subnet Mask, Default Gateway, DNS Servers
→ OK → Close
```

Windows (netsh — scriptable):

```
netsh interface ip set address name="Ethernet" static 192.168.1.100 255.255.255.0 192.168.1.1
netsh interface ip set dns name="Ethernet" static 8.8.8.8
netsh interface ip add dns name="Ethernet" 8.8.4.4 index=2
```

Windows (PowerShell):

```
# Get interface name first
Get-NetAdapter

# Remove any existing DHCP IPs
Remove-NetIPAddress -InterfaceAlias "Ethernet" -Confirm:$false

# Remove existing default route (if any)
Remove-NetRoute -InterfaceAlias "Ethernet" -DestinationPrefix "0.0.0.0/0" -Confirm:$false

# Set static IP
New-NetIPAddress -InterfaceAlias "Ethernet" -IPAddress 192.168.1.100 -PrefixLength 24 -DefaultGateway 1

# Set DNS
Set-DnsClientServerAddress -InterfaceAlias "Ethernet" -ServerAddresses ("8.8.8.8","8.8.4.4")
```

Linux (ip command — temporary, lost on reboot):

```
# Remove DHCP-assigned address
sudo ip addr flush dev eth0

# Add static address
sudo ip addr add 192.168.1.100/24 dev eth0

# Add gateway
sudo ip route add default via 192.168.1.1

# Set DNS (edit resolv.conf)
echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf
echo "nameserver 8.8.4.4" | sudo tee -a /etc/resolv.conf
```

Linux (NetworkManager — persistent):

```
# Modify the connection
nmcli connection modify "MyConnection" \
    ipv4.method manual \
    ipv4.addresses "192.168.1.100/24" \
    ipv4.gateway "192.168.1.1" \
    ipv4.dns "8.8.8.8 8.8.4.4"

# Bring it up
nmcli connection up "MyConnection"
```

Linux (/etc/network/interfaces — Debian/Ubuntu traditional):

```
auto eth0
iface eth0 inet static
    address 192.168.1.100
    netmask 255.255.255.0
    gateway 192.168.1.1
    dns-nameservers 8.8.8.8 8.8.4.4
```

```
sudo ifdown eth0 && sudo ifup eth0
```

iPhone:

```
Settings → Wi-Fi → tap (i) next to network → Configure IP → Manual
→ Enter IP, Subnet Mask, Router (gateway)
→ Configure DNS → Manual → add DNS servers
```

Method B: DHCP Reservation on the Router

Tell the router: "Whenever device with MAC `AA:BB:CC:DD:EE:FF` asks for an IP, always give it `192.168.1.100`."

The device still uses DHCP — it just always gets the same IP.

To find the device's MAC address:

Windows:

```
ipconfig /all
# Look for "Physical Address" under the relevant adapter
```

Linux:

```
ip link show       # "link/ether" field
# or
cat /sys/class/net/eth0/address
```

On router (varies by firmware, but concept is the same):

- **pfSense:** Services → DHCP Server → Add static mapping
- **OpenWRT:** Network → DHCP and DNS → Static Leases → Add
- **DD-WRT:** Services → Services → DHCP Server → Add static lease
- **Most home routers:** Look in "DHCP" settings for "Static IP Lease", "Address Reservation", or "Fixed IP"
- Enter: MAC address and desired IP

Verify it worked:

```
# Linux: release and renew
sudo dhclient -r eth0    # release
sudo dhclient eth0       # renew
ip addr show eth0        # check new IP

# Windows: release and renew
ipconfig /release
ipconfig /renew
ipconfig   # verify
```

Pros and Cons

Approach	Pros	Cons
Static on device	Works even if DHCP server is down	Must reconfigure if moved to different network; IP may conflict if DHCP pool not adjusted
DHCP reservation	Central management; device is portable; easy to change	DHCP server must be up for initial IP assignment; if device moves to another network, it may not get expected IP

Critical rule: If using static IP on the device, set the IP **outside the DHCP pool range** to prevent the DHCP server from assigning that IP to someone else. Find your DHCP range (usually shown in router settings) and pick an IP below or above it. E.g., if DHCP pool is `.100–.200`, assign static IPs in `.1–.99` or `.201–.254`.

Scenario 2: No Internet — Systematic Diagnosis

Goal: Work down from the physical layer up until you find where it breaks.

Layer 1 — Physical

```
Is the cable plugged in?
Is the link light on the NIC and switch?
Is Wi-Fi showing "connected" or just "associated"?
```

Windows:

```
Get-NetAdapter | Select-Object Name, Status, MediaConnectionState
# Status = "Up", MediaConnectionState = "Connected" = physical link OK
# Status = "Up", MediaConnectionState = "Disconnected" = no link (cable issue)
```

Linux:

```
ip link show
# Look for LOWER_UP in flags = physical link is active
# NO-CARRIER = no physical link
```

Layer 2 — Do You Have a Local Network Address?

```
ipconfig        # Windows
ip addr show    # Linux
```

If you see `169.254.x.x` (Windows APIPA) or `0.0.0.0` → DHCP failed.

Diagnose DHCP:

```
# Linux: manually run DHCP client in verbose mode
sudo dhclient -v eth0

# Windows: check event log or try release/renew
ipconfig /release
ipconfig /renew
```

DHCP failure causes:

- IP pool exhausted on the router
- MAC filtering blocking you
- MAC randomization causing a new identity each time (reservation won't match)
- DHCP server is down
- VLAN mismatch

Layer 3 — Can You Reach Your Gateway?

```
# Windows — what is my gateway?
ipconfig
# then:
ping 192.168.1.1     # (replace with your gateway IP)

# Linux
ip route show
# look for: default via x.x.x.x
ping 192.168.1.1
```

If gateway ping fails:

- Check ARP: `arp -a` — does the gateway have an entry?
- If no ARP entry: L2 problem — subnet mismatch, wrong gateway IP, gateway offline
- If ARP entry exists but ping fails: Gateway may block ICMP; try `Test-NetConnection 192.168.1.1 -Port 80` (Windows) or `nc -w 1 192.168.1.1 80` (Linux) to test TCP instead

Layer 3 — Can You Reach the Internet by IP?

```
ping 8.8.8.8    # Google's DNS server; if this works, L3 internet works
ping 1.1.1.1    # Cloudflare DNS
```

If gateway ping works but `8.8.8.8` fails:

- NAT might be broken on the router
- ISP connectivity issue
- Route to internet is missing
- Check router's WAN interface: does the router itself have internet?

Layer 7 — Does DNS Work?

```
# Windows
nslookup google.com
# or
Resolve-DnsName google.com

# Linux
dig google.com +short
# or
nslookup google.com
```

If `ping 8.8.8.8` works but `ping google.com` fails → DNS is broken.

Fix DNS test:

```
# Test with a specific DNS server directly
nslookup google.com 8.8.8.8    # Windows
dig @8.8.8.8 google.com        # Linux
```

If direct DNS server query works but configured DNS doesn't:

- Your configured DNS server is unreachable or broken
- Change DNS to `8.8.8.8` / `1.1.1.1`

Application Layer — Does HTTP/HTTPS Work?

```
# Windows
Test-NetConnection -ComputerName google.com -Port 443

# Linux
curl -v https://google.com
```

If DNS works and TCP connects but no page loads:

- TLS certificate issue
- Proxy required but not configured
- Content filtering blocking the site
- Application firewall

Complete Diagnostic Command Sequence

Windows:

```
ipconfig /all
ping 127.0.0.1
ping 192.168.1.1        (your gateway)
ping 8.8.8.8            (internet by IP)
nslookup google.com
nslookup google.com 8.8.8.8
tracert 8.8.8.8         (where does it stop?)
Test-NetConnection google.com -Port 443
```

Linux:

```
ip addr show
ip route show
ping -c 3 127.0.0.1
ping -c 3 192.168.1.1
ping -c 3 8.8.8.8
dig google.com
dig @8.8.8.8 google.com
traceroute 8.8.8.8
curl -I https://google.com
```

Scenario 3: Quick Network Between Two Windows Machines

Method A: Direct Connection (Crossover Cable or Modern Auto-MDI/X)

Modern NICs auto-negotiate crossover (Auto-MDI/X), so a straight-through Ethernet cable works between two computers now. Connect both PCs directly.

Set static IPs on both machines (DHCP won't work — no server):

PC1:

```
New-NetIPAddress -InterfaceAlias "Ethernet" -IPAddress 192.168.100.1 -PrefixLength 24
```

PC2:

```
New-NetIPAddress -InterfaceAlias "Ethernet" -IPAddress 192.168.100.2 -PrefixLength 24
```

Test:

```
# From PC1:
ping 192.168.100.2

# From PC2:
ping 192.168.100.1
```

This gives you a direct Layer 3 link. For file sharing, ensure the Windows firewall allows File and Printer Sharing, or use a quick Python HTTP server:

```
# PC1 serves files from current directory on port 8000:
python -m http.server 8000

# PC2 downloads:
curl http://192.168.100.1:8000/myfile.txt -o myfile.txt
```

Method B: Mobile Hotspot (Internet Sharing)

PC1 has internet via Ethernet; share it to PC2 via Wi-Fi.

Windows 10/11:

```
Settings → Network & Internet → Mobile Hotspot → toggle On
Set SSID and password
PC2 connects to this Wi-Fi network
```

PC2 will receive a DHCP address from the hotspot (typically `192.168.137.x`). PC1 handles NAT.

To do this via `netsh` (older method / scriptable):

```
# On PC1 (the machine with internet)
netsh wlan set hostednetwork mode=allow ssid="QuickNet" key="password123"
netsh wlan start hostednetwork
```

Then enable sharing:

```
Control Panel → Network Connections → right-click the internet-connected adapter
→ Properties → Sharing → "Allow other network users to connect through this computer's Internet connect
→ Select the virtual hosted network adapter from dropdown
```

Method C: ICS (Internet Connection Sharing) via Ethernet

```
Control Panel → Network Connections
→ Right-click the INTERNET adapter (not the one going to PC2)
→ Properties → Sharing tab
→ Check "Allow other network users to connect through this computer's Internet connection"
→ Choose the LOCAL adapter (the one connected to PC2) from the dropdown
```

PC1's local adapter will be assigned `192.168.0.1`. PC2 set to DHCP will receive an IP in `192.168.0.x`.

Scenario 4: Proxy Settings — What Do I Enter?

A screen asks for proxy settings. This happens in:

- Corporate environments (Zscaler, Symantec Web Gateway, Squid)
- Captive portal bypass
- Development tool configurations (npm, pip, apt, git)

Get Proxy Settings

From a working machine on the same network:

Windows:

```
netsh winhttp show proxy
# or
reg query "HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings" /v ProxyServer
reg query "HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings" /v AutoConfigURL
```

Linux (if already set):

```
echo $http_proxy
echo $https_proxy
env | grep -i proxy
```

From the corporate IT documentation or network admin.

Manual Proxy

Enter exactly: `proxy.company.com` and port `8080` (or whatever IT specifies). Some proxies require authentication (username/password popup appears after configuring).

Windows:

```
Settings → Network & Internet → Proxy → Manual proxy setup
Address: proxy.company.com
Port: 8080
```

Linux (environment variables — affects curl, wget, apt, pip, etc.):

```
export http_proxy="http://proxy.company.com:8080"
export https_proxy="http://proxy.company.com:8080"
export no_proxy="localhost,127.0.0.1,10.0.0.0/8,192.168.0.0/16,.company.com"

# Make permanent (bash):
echo 'export http_proxy="http://proxy.company.com:8080"' >> ~/.bashrc
echo 'export https_proxy="http://proxy.company.com:8080"' >> ~/.bashrc
```

Authenticated proxy:

```
export http_proxy="http://username:password@proxy.company.com:8080"
```

Note: password in environment variables is a security risk on shared systems.

PAC File

If given a URL like `http://proxy.company.com/proxy.pac` or `http://wpad.company.com/wpad.dat`:

Windows:

```
Settings → Network & Internet → Proxy → Automatic proxy setup
Script address: http://proxy.company.com/proxy.pac
```

Linux (GNOME):

```
Settings → Network → Network Proxy → Automatic → URL: http://proxy.company.com/proxy.pac
```

The system downloads the PAC file and executes the JavaScript `FindProxyForURL()` function for each connection to determine if/which proxy to use.

Bypass Proxy for Local Addresses

Always add local networks to the bypass list so traffic to local devices doesn't go through the proxy:

```
no_proxy: localhost, 127.0.0.1, 192.168.0.0/16, 10.0.0.0/8, *.company.com
```

Per-Tool Proxy Config

Some tools don't respect OS proxy settings:

```
# git
git config --global http.proxy http://proxy.company.com:8080
git config --global https.proxy http://proxy.company.com:8080

# npm
npm config set proxy http://proxy.company.com:8080
npm config set https-proxy http://proxy.company.com:8080

# pip
pip install somepackage --proxy http://proxy.company.com:8080
# or in ~/.pip/pip.conf:
[global]
proxy = http://proxy.company.com:8080

# apt (Debian/Ubuntu)
# /etc/apt/apt.conf.d/01proxy
Acquire::http::Proxy "http://proxy.company.com:8080";
```

Scenario 5: See What Devices Are on My Network

Windows

ARP table (only shows recently communicated devices):

```
arp -a
```

Ping sweep then ARP (find devices in subnet):

```
for /L %i in (1,1,254) do ping -n 1 -w 100 192.168.1.%i > nul
arp -a
```

This sends quick pings to all hosts in `192.168.1.x`, populating the ARP table. Then `arp -a` shows who responded. Fast but misses hosts that block ICMP.

nmap:

```
nmap -sn 192.168.1.0/24           # ping sweep (discovers live hosts)
nmap -sn 192.168.1.0/24 -oG -    # grep-friendly output
nmap -A 192.168.1.0/24            # aggressive: OS detect + port scan (slow, noisy)
nmap --open 192.168.1.0/24        # show only hosts with open ports
```

Advanced IP Scanner (Windows GUI): Free tool, fast, shows hostname + MAC + manufacturer from OUI. Useful for quick visual overview.

Linux

arp-scan (most reliable ARP-level discovery):

```
sudo arp-scan --interface=eth0 --localnet
# or specify subnet explicitly:
sudo arp-scan 192.168.1.0/24
```

Sends ARP requests to every IP in the subnet. Shows IP, MAC, and manufacturer. Much faster than nmap for local discovery. **Requires root.**

nmap ping sweep:

```
sudo nmap -sn 192.168.1.0/24        # ICMP ping + ARP (root) or just ICMP (non-root)
nmap -sn 192.168.1.0/24            # non-root: ICMP only

# Output: lists up hosts with their MAC and vendor
```

netdiscover (passive ARP monitoring):

```
sudo netdiscover -i eth0 -r 192.168.1.0/24    # active scan
sudo netdiscover -i eth0 -p                    # passive mode (just listen for ARP traffic)
```

What the Output Tells You

```
192.168.1.1    aa:bb:cc:dd:ee:ff    3   42   Cisco Systems (gateway/router)
192.168.1.50   11:22:33:44:55:66   1   60   Apple Inc.    (someone's MacBook/iPhone)
192.168.1.100  de:ad:be:ef:ca:fe   1   46   (Unknown)     (locally administered MAC — spoofed or randon
```

- **MAC OUI lookup:** First 3 bytes identify manufacturer. Helps identify device type.
- `de:ad:be:ef:ca:fe` or any MAC starting with `02:`, `06:`, `0a:`, `0e:` in the first byte (bit 1 set) → locally administered → likely randomized (phone, laptop with MAC randomization) or spoofed.
- Devices that appear in ARP scan but not in `arp -a`: They're on the network but may not have communicated with your machine.

To get hostnames from IPs:

```
# Reverse DNS lookup
nmap -sL 192.168.1.0/24    # list scan with reverse DNS
dig -x 192.168.1.1         # single reverse lookup
```

Scenario 6: Wi-Fi Keeps Dropping — Systematic Diagnosis

Step 1: Characterize the Problem

- Does it disconnect completely or just lose internet while staying associated?

- How often? Every X minutes?
- On all devices or just one?
- Associated with specific times of day (interference source that turns on/off)?
- Is the IP lost, or just connectivity?

Step 2: Check Signal

Windows:

```
netsh wlan show interfaces
# Look at: Signal (%), Receive Rate, Transmit Rate, Radio type
```

Signal % to quality:

- 80–100%: Excellent
- 60–79%: Good
- 40–59%: Fair
- Below 40%: Poor (expect drops)

Linux:

```
watch -n 1 iwconfig wlan0    # update every second
# Look at: Signal level (dBm), Link Quality, Tx excessive retries
```

Good dBm thresholds:

- Above -50 dBm: Excellent
- -50 to -60 dBm: Good
- -60 to -70 dBm: Fair (starting to see issues under load)
- Below -70 dBm: Poor (expect drops, retries)
- Below -80 dBm: Unusable

Step 3: Check for Interference

```
# Linux: scan for nearby APs and their channels
sudo iw dev wlan0 scan | grep -E "SSID:|freq:|signal:"

# Windows: see channel and network density
netsh wlan show networks mode=bssid
```

If many networks are on the same channel as yours → channel contention. Change your AP to a less-congested channel.

Check for non-Wi-Fi interference:

- Microwave ovens (2.4 GHz, 12.2–12.5 cm wavelength)
- Bluetooth (2.4 GHz, but frequency-hops — less impact)
- Cordless phones (DECT is 1.9 GHz, mostly fine; older ones on 2.4 GHz bad)
- Baby monitors (often 2.4 GHz, constant carrier)

- Switching power supplies, LED dimmers

Step 4: Check DHCP Lease

If the IP drops but you stay associated:

```
# Linux: check lease info
cat /var/lib/dhclient/dhclient.leases
# or
cat /var/lib/NetworkManager/dhclient-*.conf

# Windows: check lease expiry
ipconfig /all | findstr "Lease"
```

A lease expiring without renewal causes loss of IP (DHCP server offline, or NAT state table expired). Solution: increase lease time on router, or use DHCP reservation.

Step 5: Check for Driver Issues (Linux)

```
# Check kernel messages around disconnect time
dmesg | tail -50
# Look for: firmware errors, regulatory domain changes, channel switch announcements

# Check driver
lspci | grep -i network
lsmod | grep -i wifi

# Check firmware version
sudo ethtool -i wlan0    # driver and firmware version
```

Driver issues manifest as:

- Regular disconnects every N minutes (firmware bug with power save)
- "Deauthenticated reason 6/7" in dmesg (AP deauthing client due to inactivity or keepalive failure)

Step 6: Disable Power Save (Common Fix)

Wi-Fi power saving causes the NIC to sleep between DTIM intervals. This can cause the AP to give up on the connection.

Linux (temporary):

```
sudo iwconfig wlan0 power off
# Verify:
iwconfig wlan0 | grep "Power Management"
```

Linux (persistent via NetworkManager):

```
# /etc/NetworkManager/conf.d/wifi-powersave.conf
[connection]
wifi.powersave = 2    # 2 = disable, 3 = enable
```

```
systemctl restart NetworkManager
```

Windows: Device Manager → Network Adapters → [adapter] → Properties → Power Management → uncheck "Allow the computer to turn off this device to save power"

Step 7: Check AP Logs

If you have access to the AP admin page, check:

- Number of connected clients (overloaded AP?)
- Any deauthentication events for your device
- Error counters on the radio
- Channel utilization (above 80% causes significant performance degradation)
- Beacon interval and DTIM settings

Step 8: Persistent Monitoring

Linux — log signal strength over time:

```
while true; do
    echo "$(date): $(iwconfig wlan0 | grep 'Signal level')" >> /tmp/wifi_signal.log
    sleep 5
done
```

Watch for specific disconnect events:

```
journalctl -f | grep -i "wlan\|wifi\|dhcp\|disconnect"
```

Windows — netsh log:

```
netsh wlan show wlanreport
# Generates an HTML report in C:\ProgramData\Microsoft\Windows\WlanReport\
```

The WLAN report shows disconnect events with reasons and timestamps — very useful for correlating drops with times.